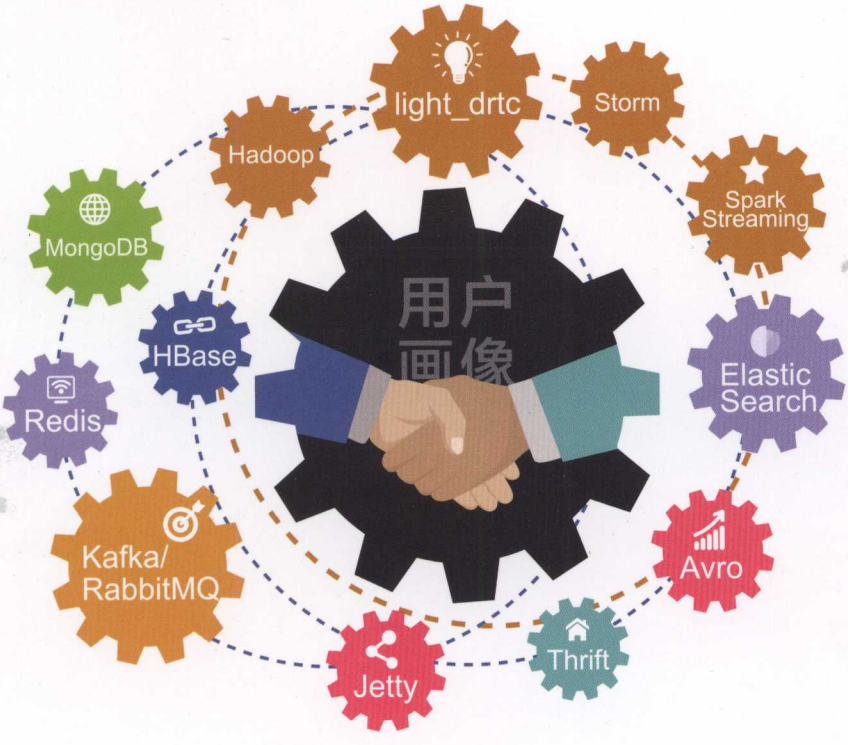
Broadview
www.broadview.com.cn

开源作者自研轻量级分布式实时计算服务框架——light_drtc，简单易用，快速实现自定义的实时计算平台，快速实现企业所需计算实时性要求比较高的业务逻辑

学通本书，做大数据时代的“心灵捕手”！



分布式实时计算框架
原理及实践案例

王成光 著

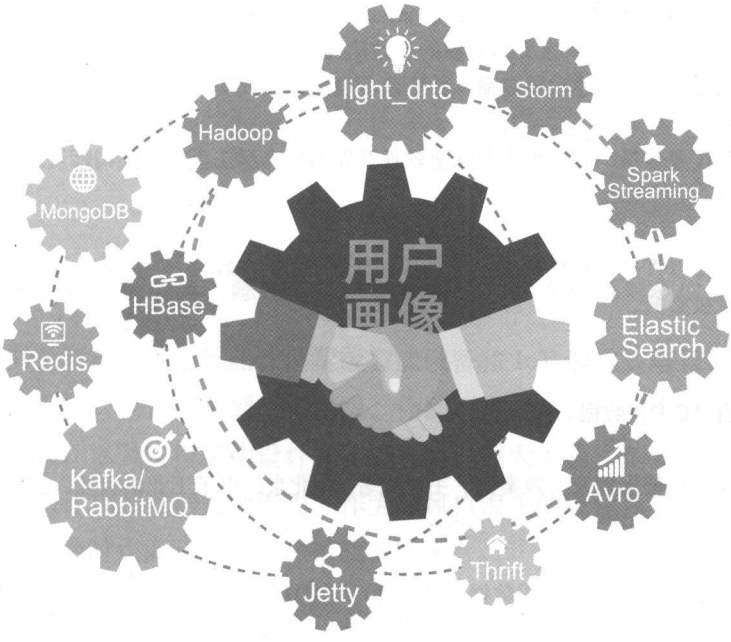
中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

关于作者

王成光

目前任职一点资讯研发架构师，硕士毕业8年，曾先后任职窝窝团、优购、搜狐、网易等架构师、技术专家职位，专注于搜索、推荐、数据挖掘领域研发工作，涉足技术范围：

- 搜索：ES / SolrCloud
- 分布式计算：Hadoop、Storm和Spark
- MQ：Kafka、RabbitMQ、ActiveMQ、ZeroMQ
- NoSQL：Reids/SSDB、Mongo3.0、HBase1.0、AeroSpike
- SOA微服务：RPC和Web Service



分布式实时计算框架
原理及实践案例

王成光 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

“授人以鱼不如授人以渔”，本书是作者以此初心写成的，主要参考当前主流分布式实时计算框架 Storm 的任务分发和 Spark Streaming 的 Mini-Batch 设计思想，以及底层实现技术，开源了作者自研的轻量级分布式实时计算框架——Light_drtc，并且重点介绍设计思想和相关实现技术（Kafka/RabbitMQ、Redis/SSDB、GuavaCache、MongoDB、HBase、ES / Solr、Thrift、Avro、Jetty），最后从工程角度向大家介绍完整的个性化推荐系统，并实例介绍 light_drtc 在用户画像实时更新的应用。本书描述浅显易懂，希望读者理解分布式实时计算的实现原理，并快速上手解决实际问题。

本书适合读者包括：高校师生及从事软件开发的中高级工程师、架构师及技术管理者等。

图书在版编目（CIP）数据

分布式实时计算框架原理及实践案例 / 王成光著. —北京：电子工业出版社，2016.9
ISBN 978-7-121-29620-8

Ⅰ. ①分… Ⅱ. ①王… Ⅲ. ①数据处理软件 Ⅳ.①TP274

中国版本图书馆 CIP 数据核字（2016）第 182107 号

责任编辑：孙学瑛
印　　刷：中国电影出版社印刷厂
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：720×1000　1/16　印张：18.5　字数：280 千字
版　　次：2016 年 9 月第 1 版
印　　次：2016 年 9 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

专家力荐

互联网连续创业者　陈超仁（原基调网络创始人、前美丽说高级副总裁）

互联网深深渗入各个行业，消费级智能设备和 IoT 爆炸式增长，联网设备数量早已突破百亿，很快将达到千亿量级。伴随而来的是计算能力需求剧增、数据呈指数级积累，以及支撑架构设计和实施能力的巨大考验。企业的竞争力将高度等同于拥有的计算及数据处理能力。作为最风口浪尖的设计及实施者，软件工程师在现今变革的时代面临的机遇和挑战前所未有。

本书作者从自身实战经验总结并研发的分布式计算框架入手，对主流支撑框架做了高度简练的介绍，并集合框架对时下热门且实用的用户画像分析这一难题进行剖析，深入浅出，将晦涩难懂的概念通过案例完全展示在读者面前，让海量数据的价值在计算魔术中逐渐显现。本书实为难得的架构实战干货，相信可以帮助工程师及架构师避免踩坑，完成进阶。

在我互联网及软件技术行业近二十年的创业经历里，在技术道路上极度追求不断突破的极客不少见，成光给我的形象却犹为深刻。记得初次见面时间不长却一见如故，交谈甚欢。这些工程师是互联网时代最精华的组成部分，他们是互联网生产力的缔造者，也一定是成功者。

阿里巴巴技术专家　赵文旭

对于解决实际问题来说，一线的经验参考最为重要。在实际实现一个业务系

统、解决一个业务问题的时候，很少会仅使用一项单一技术就可以解决，往往需要多项技术相结合使用。

在大数据处理领域，Hadoop、Storm、Spark 等核心技术是必不可少的。但是要想构建一个完整的解决方案，还需要 RPC、消息队列、缓存系统、数据库系统，一个都不能少。以上每一项技术的书籍及网上的资料不可谓不多，也不缺乏精品。但从业务解决方案入手，综合多项技术，有针对性地介绍、分析、总结的书籍却很缺乏。

本书作者王成光同学，多年来一直奋战在大数据处理领域的开发一线，具有丰富的实战经验。他曾经从无到有一手搭建了优购网（百丽电商）的搜索推荐及 BI 系统，后曾在网易等一线互联网公司任职。他在解决业务需求的基础上，逐渐沉淀提炼出一套轻量级分布式实时计算服务框架——light_drtc。本书不仅对这个自研框架的架构原理做了介绍，.同时分享了研发这套架构的心得体验。同时，对系统中应用到的各项技术进行了详细的、有针对性的介绍、对比及分析，不乏精准独到的个人见解，非常值得一读。

这本书是作者多年经验和智慧的结晶，细读它，你一定会有所收获。

TalkingData 研发副总裁　阎志涛

大数据技术发展到现在，实时流处理技术变得越来越重要。在这个大数据实时处理狂潮中，各种开源流计算框架也如雨后春笋般地涌现出来。比较常见的有 Storm、Samza、Spark Streaming、Flink，以及 Twitter 最新开源的 Heron 等等。比较遗憾的是如大部分开源技术一样，这些流计算框架都为国外公司所主导。国内虽然有 BAT 等互联网巨头，却没有在实时计算框架方面有对应自己地位的开源产品。令人稍感欣慰的是，light_drtc 作为纯粹国内技术人员主导开发的一个轻量级的流计算框架进入了开源世界，让实时计算开源框架中有了中国力量。

分布式实时计算技术对于大数据技术来讲非常重要，不过在技术上能讲解得非常透彻的中文技术书籍并不多。认识王成光时他正沉浸在开发 light_drtc 的状态中，关于分布式实时处理技术，我们聊了很多，有很多技术的见解也很一致。

令人非常高兴的是他不仅将自己实践积累的产品 light_drtc 开源出来，同时将自己多年技术积累的经验以写书的形式奉献给了广大对分布式实时计算技术有兴趣的技术人员。《分布式实时计算框架原理与实践案例》这本书不止是一本对当前实时流计算技术进行解析的一本书，同时也是作者对在工作中实战经验总结的一本书。授人以鱼不如授人以渔，我相信这本书能够给大数据技术人员，尤其是对大数据流处理技术有兴趣的人带来很大的帮助。

京东首席技术顾问　翁志

本书不仅详实地推介了作者自主研发的实时计算框架，还基本涵盖介绍了当今主流的开源计算系统。内容浅显易懂，案例切合实际，分析精准到位。对于年轻的 IT 读者来说，不失为良师益友。

中国建筑电商 CTO　邓威

本书是作者基于自己多年的思考和实践经验，经过不断提炼而得的。一方面把当前业界在实时计算领域常用的产品和技术进行深入介绍，让读者对该领域中的各个产品如何使用不再迷茫，也为具有一定经验的从业者对下一步系统中关键服务如何选型和优化提供了新的方向；另一方面作者博采众家之长而创造的 light_drtc，降低了中小企业在分布式实时计算领域的门槛，使大数据处理能力触手可得。最后通过作者的实例，在了解大数据技术如何应用的同时，也能了解到在实例背后作者所体现出的思路、方法和方案。希望通过本书能让更多的人了解大数据处理，有更多的企业挖掘出自身的数据价值。

腾讯技术副总监　鞠奇

成光对于架构的不断钻研和踏实肯干给我留下了很深的印象，非常有幸见证了他这套分布式实时计算系统在新闻推荐领域中的应用。这本书结合他多年的一线实践经验，详细阐述了分布式计算系统、搜索架构和数据库等在企业应用的经

历，对于初学者和想深入理解这一方面知识的同学会起到很好的引导。感谢成光为国内的工程架构贡献自己的一份力量！

大码美衣 CTO　王伟涛

作者硕士从事计算机应用中文信息处理的研究工作，毕业后在百丽、好乐买、搜狐和网易等担任搜索、推荐架构师等关键岗位。多年来一直从事大数据相关的开发和研究工作，从未停止对技术的了解和钻研。作者结合当前主流开源软件的特点，以及中小型企业人才和资源不足的困难，利用多年来的技术积累和沉淀，独力研发了一套轻量级分布式实时计算服务框架——Light_drtc，包含实时数据收集服务、资源协调及任务管理服务和任务计算服务，可以帮助企业快速搭建自定义的实时计算平台，让企业聚焦于业务数据的分析和处理。

本书系统介绍了 Light_drtc 的设计思想、功能,以及核心技术。同时深入浅出地介绍了当前主流开源大数据处理技术，如 Hadoop，Spark 及 Storm 等主流计算框架,以及消息队列、内存数据库、NoSQL、搜索、RPC 框架等核心技术架构，帮助开发者系统全面地了解大数据平台的核心技术。

作为中小型企业的技术负责人，对此书的作者表示感谢，作者提供了一套轻量级的解决方案，并介绍了相关的技术环节，让小企业可以专注于数据的应用，而不是花大量的时间用于搭建平台，切实提供了很大的帮助。

感谢作者花了大量精力对中小企业的支持以及对开源的支持，希望本书可以帮助更多的企业，也为国家大数据平台的发展贡献一份力量。

前　言

“授人以鱼不如授人以渔。”——语出《淮南子 · 说林训》，道理很简单，鱼是目的，钓鱼是手段，一条鱼能解一时之饥，却不能解长久之饥，如果想永远有鱼吃，那就要学会钓鱼的方法。

随着时代发展，互联网，尤其是移动互联网的全民普及趋势，任何一个行业的相关产品都会有很多表示用户兴趣点的行为数据，像用户的浏览、收藏、分享、购买、评论、点赞和搜索等行为由此构建出海量用户行为数据。如何快速有效地使用上述大数据，挖掘出用户对产品的兴趣点，实时更新用户画像，进而向用户推荐其当前最感兴趣的产品及广告，是目前众企业所普遍关心的问题，也是大数据的价值所在。

大数据是时代产物，除了 BAT、网易、搜狐、新浪、京东等一线互联网企业需要相关处理，国内更多中小企业或者传统行业的巨无霸也都需要大数据处理技术。一线互联网企业由于自己本身就做互联网业务拥有人才优势，处理大数据相对简单。但中小企业及传统行业巨无霸在面对自己日积月累的大数据时困难重重，主要是本身没有处理能力，也没有相关硬件设施支持。

目前大数据处理技术，以开源界大名鼎鼎的 Hadoop、Spark、Storm 及后起之秀 Flink 为代表，当然国内一线互联网企业像 BAT 也都有各自独特的处理技术，但由于国内大环境所致，商业公司在运作时首先考虑的是商业及保密措施，使得国内开源界在分布式处理方面相对薄弱，基本上是空白战场。

目前看来，Hadoop 比较适合用于离线数据处理，Spark 及 Storm 的实时处理

技术正好弥补了 Hadoop 实时处理的欠缺。虽然 Hadoop、Spark 及 Storm 都在快速发展，但国内真正深入理解这三者的人毕竟凤毛麟角，而且 Hadoop 集群本身动辄就需要上百台服务器的集群，这对中小企业来说基本上是不可能的，而且中小企业也很少能够拥有精通 Hadoop、Spark 及 Storm 的资深技术人员，这就导致绝大部分中小企业的大数据处理基本上处于停滞状态，它们眼睁睁地看着大量数据产生，而无法进行相应处理。即使众中小企业花费巨资招聘了 Storm、Spark Streaming 开发人员，由于自身平台的限制，开发者基本上只是简单地调用其封装好的 API，很难从源码跟踪到问题根源，也就造成生产环境下有很多问题难以解决。

时代在变，技术也在变，没有任何一项技术会解决所有问题，对企业及个人开发而言，适合自己的才是最好的。Hadoop3.x 以后将会调整方案架构，将 MapReduce 基于内存+IO+磁盘，共同处理数据，这点和当前的 Spark 很像；Storm1.0 也打破了之前一直存在的诟病：Nimbus 单点问题，实现了 HA，这点和阿里推出的 JStorm 很像，而 JStorm 本身也是源自 Storm；最近 Twitter 又推出 Heron，兼容 Storm。Storm 也提供了类似 Spark Streaming 的微批处理方式——trident（以一组 tuple 为单位）。上述开源项目之间都在互相学习对方优点，取其精华为己所用。

鉴于上述原因，本书作者经过多年的深入思考，结合自己硕士毕业 8 年多的一线互联网开发经验，根据 Hadoop 的 Map/Reduce 及当前主流实时计算框架 Storm 的任务分发和 Spark Streaming 的 Mini-Batch 处理思想，利用时下比较流行的 MQ、RPC、NoSQL 等，独力研发了一套**轻量级分布式实时计算服务框架——light_drtc**。其最大特点就是简单易用，它可以帮助开发者快速实现自定义的实时计算平台，其设计目的是为了降低当前大数据时代的分布式实时计算入门门槛，方便初、中级读者上手，快速实现企业所需计算实时性要求比较高的业务逻辑。它本身既可以作为独立的分布式实时计算平台，也可以以嵌入式方式，作为其他项目的基础类库存在。

Light_drtc 目前以 Java8 为基础设计和实现，框架主要包括三部分：实时数据收集服务（CollectNode，简称 CN）、资源协调及任务管理服务（AdminNode，简称 AN）和任务计算服务（JobNode，简称 JN）。这套框架扩展灵活，各个相关组件可以自由扩展（目前框架已整合 Kafka 和 RabbitMQ，物理分布上可以将 CN 和

AN 整合在一起），集群节点所完成计算所需要的开发语言不仅限于 Java，也可以用时下流行的各种开发语言。

对中小企业而言，利用 light_drtc 搭建分布式实时计算平台，不需要庞大的服务器集群规模，完全可以根据自己业务需要。例如抽取 9 台服务器：其中 3 台服务器同时兼做 CN 和 AN，6 台服务器做 JN，每个 AN 独立管理 2 个 JN，即可搭建自己的高可用分布式实时计算集群。在 light_drtc 框架中，每个 AN 都有自己独立管理的 JN，且每个 AN 至少独立管理 1 个 JN。框架使用比较方便，尤其是如果开发者也选用 Kafka 或 RabbitMQ，对于 CN 和 AN 而言，仅需要开发者自定义 MQ 相关配置及实现框架所定义的流数据解析接口，将实现类传递给框架 AN 节点启动类即可。对于 JN，则需要开发者按照自己业务需求自行实现，框架中有丰富的实例可供参考。

本书偏重工程架构方面，主要内容除了作者自研的 light_drtc 详细介绍，还会以作者多年一线互联网开发经验角度，陆续向读者介绍当前主流 MQ、NoSQL、全文检索 Elasticsearch/Solr，及常用微服务架构技术实现 RPC 和 Web Service 的多种框架，最后介绍整个新闻个性化推荐系统的各个组成部分，对核心模块用户画像实时更新做了详细设计，并对比 Storm、Spark Streaming 和 light-drtc 不同实现方式。作者希望读者通过阅读本书，让您对分布式实时计算系统的设计原理及相关实现技术有更加清晰的理解，也希望让众多中小企业可以快速组建自己的分布式实时计算平台，也同时为国内分布式处理技术贡献一点自己的力量。

最后给读者朋友分享一下个人多年学习的一点心得：万丈高楼平地起，任何一项别人看似游刃有余的技能都是经过时间打磨后的熟能生巧，希望读者朋友们经过自己的奋斗都可以实现人生目标，达到人生顶峰。

王成光

2016/8/5

目　录

1 第1章 分布式实时计算框架介绍

目前分布式计算框架以大名鼎鼎的 Hadoop Map/Reduce、Spark Streaming 和 Storm 为代表，可以说 Hadoop 奠定了最近 10 年来开源分布式计算框架的基石。

1.1 分布式计算 Hadoop

Hadoop 是原 Yahoo 的资深技术专家 Doug Cutting 根据 Google 发布的学术论文研究而来的。Hadoop 是一个能够对大量数据进行分布式处理的软件框架，它以一种可靠、高效、可伸缩的方式进行数据处理。

Hadoop2.0 框架最核心的设计就是：HDFS、MapReduce 和 Yarn。HDFS 是一个分布式文件系统，为海量数据提供了存储；MapReduce 是一个离线处理框架，由编程模型（新旧 API）、运行时环境（JobTracker 和 TaskTracker）和数据处理引擎（MapTask 和 ReduceTask）三部分组成，为海量数据提供了计算框架；Yarn 是一个框架管理器，为各种框架进行资源分配和提供运行时环境。MRv2 则是运行在 YARN 之上的第一个计算框架。

Hadoop 2.0 中对 HDFS 进行了改进，使 NameNode 可以横向扩展成多个，其中，每个 NameNode 分管一部分目录，这不仅增强了 HDFS 的扩展性，也使 HDFS

具备了隔离性。利用 Hadoop 进行分布计算的核心是如何优化分解一个大计算任务为若干独立子任务，从而并行运行。作者自研分布式计算框架在 AN 的资源管理和任务调度方面也是参考了这点。

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台。用户可以轻松地在 Hadoop 上开发和运行处理海量数据的应用程序。它主要有以下几个优点：

（1）高可靠性：Hadoop 按位存储和处理数据的能力值得人们信赖。

（2）高扩展性：Hadoop 是在可用的计算机集簇间分配数据并完成计算任务，这些集簇可以方便地扩展到数以千计的计算节点中。

（3）高效性：Hadoop 能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。

（4）高容错性：Hadoop 能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。

（5）低成本：与一体机、商用数据仓库及 QlikView、Yonghong Z-Suite 等数据集市相比，Hadoop 是开源的，项目的软件成本因此会大大降低。

即将推出的 Hadoop3.0 将会调整方案架构，将 Mapreduce 基于内存+IO+磁盘，共同处理数据，其实最大改变的是 HDFS，HDFS 通过最近 black 块计算，根据最近计算原则，本地 black 块，加入到内存，先计算，通过 IO，共享内存计算区域，最后快速形成计算结果。虽然性能改善让人很期待，不过距离使用还有一段时间，目前尚不能应用到实际生产环境。

诚然，Hadoop 在分布式计算框架方面的贡献无人可出其右，但它也不是万能的，Hadoop 适合离线批处理计算任务，而且为了凸显 Hadoop 集群优势，业内一般都是几百台甚至上千台服务器集群规模。对于中小企业而言，一般不会组建这么大的计算集群，反而更需要一个小而精的分布式计算平台。此外，Hadoop 一般只用来离线批处理计算，因为它必须将所有输入数据都处理完才返回最终计算结果，这对于实时性要求比较高的在线学习显然不合适。

1.2 分布式实时计算

随着大数据的发展，人们对大数据的处理要求也越来越高，原有的批处理框架 MapReduce 适合离线计算，却无法满足实时性要求较高的业务，如实时推荐、用户行为实时分析等。与离线计算相比，目前企业对于实时计算的需求更迫切，因为用户兴趣点本身就是一个不确定因素，只有紧紧抓住用户当前喜好，据此进行重点相关推荐，才会牢牢抓住用户。当前分布式实时计算框架以 Spark、Storm 和 Flink 为代表。

1.2.1 Spark Streaming

1. 基本介绍

Apache Spark 是加州大学伯克利分校的 AMPLabs 开发的开源分布式轻量级通用计算框架。Spark 是一个构建在 Hadoop 基础之上类似于 MapReduce 的分布式计算框架，其核心是弹性分布式数据集（RDD），提供了比 MapReduce 更丰富的模型，可以在内存中对数据集进行多次快速迭代，以支持复杂的数据挖掘算法和图形计算算法。

Spark Streaming 是 Spark 核心 API 的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。其原理是将 Stream 数据流依据滑动时间窗口切分成一个个小的时间间隔（比如几秒）的独立数据集，即将其离散化并转换成一个个数据集（RDD），然后分批处理处理这些小的 RDD。

Spark Streaming 的数据源主要有两类：外部文件系统，如 HDFS，Streaming 可以监控一个目录中新产生的数据，并及时处理。如果出现 fail，可以通过重新读取数据来恢复，绝对不会有数据丢失；网络系统：如 MQ 系统（Kafka、ZeroMQ、Flume 等）。

Spark Streaming 默认会在两个不同节点加载数据到内存，一个节点 fail 了，系统可以通过另一个节点执行数据重算。假设正在运行 InputReceiver 的节点 fail 了，则可能会丢失一部分数据。

2. 计算流程

图 1.1 显示了 Spark Streaming 的整个计算流程[1]，Spark Streaming 是将流式计算分解成一系列短小的批处理作业。

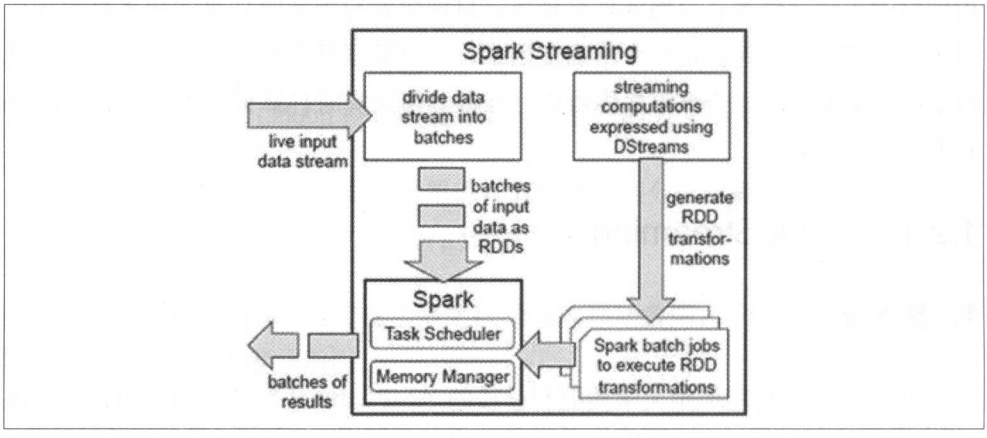


图 1.1 Spark Streaming 构架

这里的批处理引擎是 Spark Core，也就是把 Spark Streaming 的输入数据按照 batch size（如 1 秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成 Spark 中的 RDD（Resilient Distributed Dataset），然后将 Spark Streaming 中对 DStream 的 Transformation 操作变为针对 Spark 中对 RDD 的 Transformation 操作，将 RDD 经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加或者存储到外部设备。

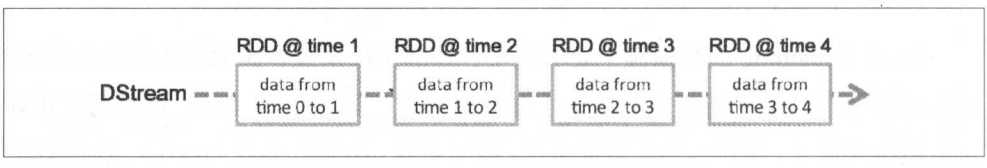


图 1.2 Spark Streaming DStream 结构

如图 1.2 所示，在 Spark Streaming 中，DStream（Discretized Stream）作为

1 实时流计算 Spark Streaming 原理介绍 http://m635674608.iteye.com/blog/2248368

Spark Streaming 的基础抽象，它代表持续性的数据流。这些数据流既可以通过外部输入源赖获取，也可以通过现有的 Dstream 的 transformation 操作来获得。在内部实现上，如图 1.2 所示，在 Spark Streaming 中，DStream 由一组时间序列上连续的 RDD 来表示，每个 RDD 都包含了自己特定时间间隔内的数据流，所以 DStream 就是一个个 batch 的有序序列，时间是连续的，按照时间间隔将数据流分割成一个个离散的 RDD 数据集。

3．内部核心操作

Spark 支持两种操作类型：Transformation 和 Action。Transformation 从一个已知的 RDD 数据集经过转换得到一个新的 RDD 数据集，这些 Transformation 操作包括 map、filter、flatMap、union、join 等，而且 Transformation 具有 lazy 的特性，调用这些操作并没有立刻执行对已知 RDD 数据集的计算操作，而只是在调用了另一类型的 Action 操作后才会真正地执行。Action 会真正地对 RDD 数据集进行操作，给 Driver 程序返回一个计算结果，或者不返回结果，如将计算结果数据进行持久化，Action 操作包括 reduceByKey、count、foreach、collect 等。

同样 Spark Streaming 提供了类似 Spark 的两种操作类型，分别为 Transformation 和 Output 操作，它们的操作对象是 DStream，作用也和 Spark 类似：Transformation 从一个已知的 DStream 经过转换得到一个新的 DStream，而且 Spark Streaming 还额外增加了一类针对 Window 的操作，当然它也是 Transformation，但是可以更灵活地控制 DStream 的大小（时间间隔大小、数据元素个数），例如 window(windowLength, slideInterval)、countByWindow(windowLength, slideInterval)、reduceByWindow(func, windowLength, slideInterval)等。Spark Streaming 的 Output 操作允许我们将 DStream 数据输出到一个外部的存储系统，如数据库或文件系统等，执行 Output 操作类似执行 Spark 的 Action 操作，使得该操作之前 lazy 的 Transformation 操作序列能够真正地执行。

4．不足之处

Spark 被称为内存版的 Hadoop，近来也日益活跃，使用也越来越广，它赖以

日益流行和广为使用的两个核心组件就是 RDD 和 Streaming，而且它本身依托于 Hadoop。Spark Streaming 诚然有很多优点，但也有一些不足：

（1）Spark 本身构建在 Hadoop 之上，不能独立存在。

（2）Spark Streaming 处理 Mini-Batch 任务的前提是用户自行实现实时数据流的接收，并没有提供直接解决方案。

（3）Spark 本身尚处于幼年阶段，正在高速发展期，本身还存在很多问题，比如经常遇到的内存泄露问题就会让用户无从下手。

（4）Spark 框架越来越复杂，对中小企业而言，要想大规模使用，目前看基本不现实。

（5）Spark 核心组件 RDD 本身通用性层面不如 JSON 更方便，RDD 本身完全可用更加成熟的 Redis3.0 替代。

1.2.2 Storm

Apache Storm 是一个开源的专门用于事件流的分布式实时计算框架，其诞生可以追溯到当初由 BackType 公司开发的项目——这家市场营销情报企业于 2011 年被 Twitter 所收购。Twitter 旋即将该项目转为开源并推向 GitHub 平台，不过 Storm 最终还是加入了 Apache 孵化器计划并于 2014 年 9 月正式成为 Apache 旗下的顶级项目之一。Storm 有很多应用场景，包括实时数据分析、联机学习、持续计算、分布式 RPC、ETL 等。Storm 速度非常快，一个测试在单节点上实现每秒一百万的组处理。

Storm 简化了流数据的可靠处理，像 Hadoop 一样实现实时批处理。Storm 项目主要利用 Clojure 编写而成，且既定设计目标在于支持将“流”（例如输入流）与“栓”（即处理与输出模块）结合在一起，并构成一套有向无环图（简称 DAG）拓扑结构。Storm 的拓扑结构运行在集群之上，而 Storm 调度程序则根据具体拓扑配置将处理任务分发给集群当中的各个工作节点。

大家可以将拓扑结构大致视为 MapReduce 在 Hadoop 当中所扮演的角色，只不过 Storm 的关注重点放在了实时、以流为基础的处理机制身上，因此其拓扑结构默认永远运行或者直到手动中止。一旦拓扑流程启动，挟带着数据的流就会不断涌入系统并将数据交付给 Spout(而数据仍将在各 Bolt 之间循着流程继续传递)，而这也正是整个计算任务的主要实现方式。随着处理流程的推进，一个或者多个 Bolt 会把数据写到数据库或者文件系统当中，并向另一套外部系统发出消息或者将处理获得的计算结果提供给用户。Storm 框架流程[2]如图 1.3 所示。

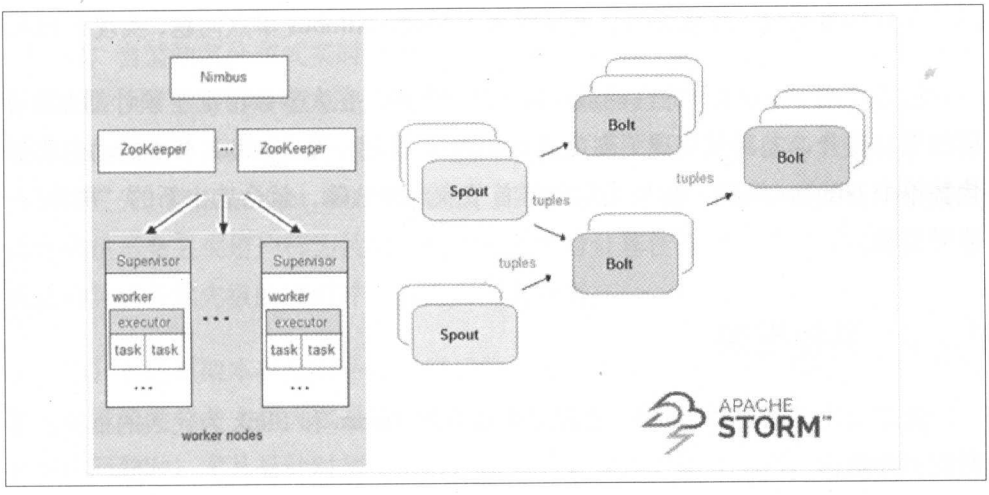


图 1.3 Storm 框架流程

在 Storm 中，先要设计一个用于实时计算的图状结构，我们称之为拓扑(topology)。这个拓扑将会被提交给集群，由集群中的主控节点（master node）分发代码，把任务分配给工作节点（worker node）执行。一个拓扑中包括 spout 和 bolt 两种角色，其中 spout 发送消息，负责将数据流以 tuple 元组的形式发送出去；而 bolt 则负责转换这些数据流，在 bolt 中可以完成计算、过滤等操作，bolt 自身也可以将数据随机发送给其他 bolt。由 spout 发射出的 tuple 是不可变数组，对应着固定的键值对。

Storm 集群非常类似 Hadoop 集群。Hadoop 上运行的是 MapReduce jobs，而

2 流式大数据处理框架 http://www.csdn.net/article/2015-03-09/2824135

Storm 运行的是 topologies。Jobs 和 topologies 本身是不同的，其中一个最大的不同就是，MapReduce job 最终会结束，而 topology 则会持续处理消息（直到你杀掉它）。Storm 集群由一个主节点和多个工作节点组成：master 节点和 worker 节点。master 节点运行一个守护进程，叫 Nimbus，类似 Hadoop 中的 JobTracker。Nimbus 负责在集群中分发代码，分配任务，以及故障检测。

阿里巴巴推出的Jstorm可以看成是Storm的Java增强版本，除了内核用纯Java实现外，还包括了 Thrift、Python、facet ui。从架构上看，其本质是一个基于 zk 的分布式调度系统，其最大的亮点是解决了 Storm Nimbus 单点问题，实现了 HA。

官方介绍 Storm 1.0 的性能最高提升了 16 倍，在大多数情况下预计都会有 3 倍的性能提升，尤其是实现了高可用 Nimbus：使用一个 Nimbus 节点的动态集群代替单个 Nimbus 实例，如果当前的群首节点出现故障，就会选出新的“群首”，值得关注。

1.2.3 其他框架

除了上述 Spark 和 Storm，后起之秀还有以 Samza 和 Flink 为代表的框架，下面逐一介绍。

Flink 是可扩展的批处理和流式数据处理的数据处理平台，设计思想主要来源于 Hadoop、MPP 数据库、流式计算系统等，支持增量迭代计算。

Apache Samza 是一个开源分布式流处理框架，专用于实时数据的处理，非常像 Twitter 的流处理系统 Storm，不同的是 Samza 基于 Hadoop，处理完毕后把结果返回 Kafka，可以帮助开发者进行高速消息处理的同时具有良好的容错能力。

1.3 为什么自研

本书将向读者介绍作者独立研发的轻量级分布式实时计算框架 Light_drtc，框架设计原理来自 Hadoop 的 map/reduce 及当前主流实时计算框架 Storm 的任务分

发和 Spark Streaming 的 Mini-Batch 处理思想。时下开源框架的确很多，这里之所以还要自研，有以下几点原因：

（1）light_drtc 框架旨在更好地解决当今电商、门户网站及广告计算的用户画像实时更新或相关实时统计，比如单位时间内（比如最近 10 秒）UV 和 PV 及 CTR 等。前述开源框架都是通用型的，由于现实业务千变万化，需求繁多，上述框架的易用性和通用性在解决实际问题时常常捉襟见肘，即使勉强使用，也会有“杀鸡使用牛刀”之感。

（2）当前知名分布式实时计算框架 Storm、Spark Streaming 在计算任务高度密集时，由于服务器资源总是有限度，所以往往会导致计算任务堆积，一旦堆积过多，往往会导致系统负载过大，系统出现异常造成很多计算任务失败，用户行为数据丢失。这点在 light_drtc 中对此做了比较巧妙的设计，会尽量保证即使流量高峰期也不会丢失用户行为数据，因为框架中对计算任务延迟做了专门处理，在保证计算任务优先级的同时并不会丢失行为数据。

（3）上述框架本身都有一定入门门槛，想精通其中任何一个都需要一定时间积累，框架的相关文档大多是英文版，对国内用户而言，使用感觉不是很亲切，在处理问题时，尤其是针对初学用户，感觉只是学会了几个 API，内部原理仍然是个黑盒，这样很难让用户真正掌握。此外框架本身也有一定缺陷，比如 Storm，如果某个 task 有问题，则可能导致整个 worker 进程崩溃，导致其他 task 也不能正常工作了，也就是说局部的错误会被放大。

（4）上述框架底层所用相关技术，往往陈旧，比如，Storm 底层的 RPC 通信使用的 Thrift，远不是最新版的 Version0.9.3，而是不到 0.8，上述框架一旦定型，底层框架所依赖技术很难与时俱进，这也会影响框架使用性能。

（5）上述开源框架都是计算平台型框架，往往开发人员开发完一个具体项目，将代码打包后提交到平台后，如果出了问题，想追踪问题根源并不是那么容易，而 light_drtc 则可以将整个代码作为开发者业务需要的基础类库，嵌入式，和业务结合可以更紧密，一旦出问题，更容易定位，从而更快解决问题。

(6) 时代在变，技术也在变，没有任何一项技术会解决所有问题。hadoop3.x 以后将会调整方案架构，将 Mapreduce 基于内存+IO+磁盘，共同处理数据，这点和当前 Spark 很像；Strom1.0 也打破了之前一直存在的诟病：Nimbus 单点问题，实现了 HA，这点和阿里推出的 JStorm 很像，而 JStrom 本身也是源自 Storm；最近 Twitter 油推出 Heron，兼容 Storm。Storm 也提供了类似 Spark Streaming 的微批处理方式——trident（一组 tuple 为单位）。上述开源项目之间都在互相学习对方优点，取其精华为己所用。作为一名互联网行业资深技术人员，完全可以综合比较当前主流的各种框架技术，根据其设计思想和实现原理及使用感悟，设计一套更适合自己使用的框架技术，作为自己核心技术，并根据最新技术进展，持续加以优化，这样在解决实际问题时，才会更加得心应手，适合自己的才是最好的，对企业而言同理。

(7) 当前分布式实时计算框架，绝大部分都是国外同仁们的杰作，由于国内氛围，即使有能力设计开发出一套不亚于前述框架的公司，出于保密及技术壁垒考虑，也都往往选择不开源。即使有一些开源的项目，像 JStorm 也是在 Storm 框架基础上的改进。所以，作者本人也是想尽自己微薄之力，为国内技术开源做点贡献，作者将自己的核心技术框架开源，目的也是希望有更多感兴趣的朋友一起参与，支持国产软件，国人当自强。

1.4 总结

本章首先向读者介绍了当前流行的、广为使用的开源分布式计算框架 Hadoop，重点介绍了分布式实时计算框架 Storm 和 Spark Streaming 实现原理，总结了各自优缺点，并向读者介绍作者自己多年的核心技术——**轻量级分布式实时计算框架 light_drtc（已开源）**，重点给读者介绍了自研原因，个中原因，相信读者可以体会到。

接下来两章，将会从框架实例使用说明及设计原理和架构等向读者介绍 light_drtc 框架。

2 第2章 light_drtc 简介及使用说明

本章主要向读者介绍作者本人所独立研发的一套轻量级分布式实时计算框架 light_drtc，重点介绍 light_drtc 框架的基本组织结构和使用说明，让有开发经验的读者朋友快速上手搭建一个属于自己的分布式实时计算平台。下章将会给大家介绍 light_drtc 框架的设计思想、技术架构，以及核心技术实现等。

2.1 light_drtc 框架简介

light_drtc 是一款遵循 Apache 协议的轻量级分布式实时计算的开源框架，它可以帮助你快速实现自定义的分布式实时计算平台。它主要参考 Hadoop 的 Map / Reduce 处理思想，和当前主流实时计算框架 Storm 的任务分发及 Spark Streaming 的 Mini-Batch 处理思想设计，目的是为了降低当前大数据时代的分布式实时计算入门门槛，方便初中级学者快速上手，实现企业所需计算实时性要求比较高的业务逻辑。它本身可以作为独立的分布式实时计算平台存在，也可以作为其他项目的基础类库，嵌入其中存在。

light_drtc 整个项目可以分为 3 部分：实时数据收集 CollectNode（CN）、任务管理 AdminNode（AN）和任务计算 JobNode（JN）三部分，三者结合，共同完成

完整的分布式实时计算系统。

目前该项目已开源，GitHub 网址为：https://github.com/lrtdc/light_drtc，开源中国网址为：http://www.oschina.net/p/light-rtdc，欢迎更多感兴趣的同学一起参与！

2.2 light_drtc 代码结构

light_drtc 框架代码结构如图 2.1 所示。

图 2.1 light_drtc 代码结构

下面详细加以说明。

“src/main/java”：框架主体包，其下相关包说明如下。

- “org.light.rtc.admin”：AN 启动核心包，分别展示了独立启动，以及分别和 Kafka、RabbitMQ 整合的启动入口；
- “org.light.rtc.api”：框架底层 RPC 相关 API 定义，使用 Thrift 自动生成。
- “org.light.rtc.base”：开发者使用框架时，自定义实现业务逻辑所需要实现

的接口或继承的基类。

- “org.light.rtc.client”：框架底层通信时内部封装的各种 API，供 CN、AN 和 JN 间的相互调用。
- “org.light.rtc.dao”：后期项目需要，框架可以扩展各种 DAO，目前仅有一个 Redis 使用方式，后续可以根据需要自行扩展增加 Mongo3.0.x 和 ES2.x 等 DAO 实例，方便开发者直接使用。
- “org.light.rtc.job”：JN 启动核心包，JN 启动入口。
- “org.light.rtc.mq”：CN 启动入口，可以单独存在，也可以和 AN 整合（目前只支持 Kafka 和 RabbitMQ）。
- “org.light.rtc.service”：AN 和 JN 服务启动时所依赖的底层服务实现。
- “org.light.rtc.timer”：CN 和 AN 资源调度器，调控 MQ 实时数据分派、提交 CN 本地队列数据给 AN 的间隔周期、AN 中计算任务分派、计算任务间隔周期、保证流量高峰期的延迟任务计算优先级、失败计算任务的重新计算，以及 AN 中对 JN 的心跳检测等。
- “org.light.rtc.util”：框架底层公共工具类包。
- “src/main/resources”：框架主体包所需配置文件。

“src/test/java”：框架测试包，也是框架三个核心部分之一，作为一个整体，完成的一个分布式实时计算系统的实例说明。其下相关包说明如下：

- “org.light.ldrtc.jober”：JN 接入实现范例，展示框架 JN 接入接口实例，并给出使用 Fork/Join 实现 JN 业务逻辑实例，如果单个计算任务逻辑流程比较复杂，这里作者建议读者考虑使用 JDK8 新特性 CompletableFuture，它可以帮助你异步完成多个并行逻辑，提高计算效率。
- “org.light.ldrtc.parser”：CN 中，完成 MQ 数据解析应用主体，并在 AN 启动前传递给它。
- “org.light.ldrtc.test”：框架使用范例，分别展示了 CN、AN 和 JN 启动实例，当然也有 CN 和 AN 整合为一体统一启动实例。
- “src/test/resources”：框架测试包所需配置文件。

2.3 light_drtc 重要配置项

light_drtc 有很多配置项，这里重点给大家介绍重要配置项，如表 2.1 所示。

表 2.1 light_drtc 重要配置项

配置项	说 明
kfZkServers	对于 Zookeeper 集群的指定，可以是多个 hostname1:port1, hostname2 :port2,hostname3:port3 必须和 broker 使用同样的 zk 配置
kfGroupId	Kafka 消费群组
kfAutoOffsetReset	当 Zookeeper 中没有初始的 offset 时候的处理方式。smallest：重置为最小值 largest：重置为最大值 anything else：抛出异常
kfTopic	设置的 Kafka 消息主题
mq.host	RabbitMQ 服务器 IP
mq.port	RabbitMQ 服务端口
mq.user	RabbitMQ 用户
mq.pswd	RabbitMQ 密码
mq.vhost	RabbitMQ 虚拟机路径
mq.exchange	RabbitMQ 交换机
mq.clickRoutKey	RabbitMQ 路由 key
mq.clickQueue1	绑定同一个路由 key 的队列 1
mq.clickQueue2	绑定同一个路由 key 的队列 2
mqDoBatchTimer	实时收集的数据流所隔秒数批量提交给任务管理节点
rtcPeriodSeconds	任务管理节点中所隔秒数将所收集的数据统分发给任务计算节点
atomJobBatchNum	任务计算节点在单位时间内所处理的按主键聚合的最大数据条数
minJobBatchNum	任务计算节点在单位时间内所处理的按主键聚合的最小数据条数
adminNodeHosts	至少使用 2 个任务管理节点，即多主模式
adminNodePort	任务管理节点启动服务所在端口，可自定义
jobNodeHosts	建议每个任务管理节点至少 3 个任务计算节点实例
jobNodePort	任务计算节点启动服务所在端口，可自定义
delayTaskDir	延迟的任务按主键聚合的 json 数据列表，按时间片段命名写入指定目录，可自定义
delayTaskFileSurfix	延迟执行的元数据文件后缀命名
maxDelayTaskNum	任务管理节点能接收的延迟任务元数据组最大个数，超限，新的元数据组将以一个文件写入指定目录

表 2.1 展示的即为框架中的核心配置属性说明，开发者配置好这些基本属性，就可以快速搭建一套适合自己业务需要的实时计算平台。值得注意的是：由于每

个AN都有自己独立的JN范围，上述配置项在实际上都需要根据实际做相应修改。比如现在服务器配置比较高，可以在一个服务器上配置多个 AN，或者一个服务器上将 CN 和 AN 合在一起部署，只要端口区分开即可。

2.4 light_drtc 和 Storm 比较

当前 Storm 也是特别活跃，这里就自研框架和 Storm 做一个全面对比，如表 2.2 所示。

表 2.2 light_drtc 和 Storm 对比

比较项	light_drtc	Storm
服务方式	目前嵌入式，作为开发库存在今后也会平台化	计算平台
应用方式	AN、JN 和 CN 组成一个整体应用	1 个 Topology 对应 1 个应用
任务分配资源调度	AdminNodeRun 运行在 AN 实现实时数据流解析接口 自实现协调	Nimbus 运行在 Master 依靠 Zookeeper 协调
计算节点	JoberNodeRun 运行在 JN 实现业务需求，计算落地	Supervisor 运行在 Worker worker 运行 topology
数据收集	继承 MqConsumer，运行在 CN 自实现 MQ 接收，每条消息数据调用 mqTimer.parseMqText(id,logText)	集成了 Kafka，可以直接在 spout 中接收 MQ 消息
HA	多主模式，更安全	单点问题
服务监控	每个服务进程都要自己实现服务监测	每个 Worker 上有一个 supervisor
任务处理	借鉴 SparkStreaming 的 Mini-batch	即可单条，也可 Mini-batch

当然，这里只是给出了两者的一个简单比较，Storm 作为当前开源界分布式实时计算的一个主流，里面有很多优秀的设计方式，而且目前也相对成熟，目前 light_drtc 向 Storm 学习和借鉴的地方还有很多，这里也只是一个粗浅的简单对比，仅供参考。

Storm 自身还有一些问题，比如计算任务异常往往会过分放大，即使单节点问题也会导致整个应用停滞。要想实现 Mini-batch 尚需开发者自行开发。

2.5 light_drtc 使用说明

light_drtc 目前尚在发展期，对于开发者而言，笔者建议大家把 light_drtc 作为基础类库模式使用方式。light_drtc 框架本身包含三个紧密关联的部分：CN、AN 和 JN，依据开发者在开发时，CN 中的 MQ 技术选型是否选用框架所支持的 Kafka 或 RabbitMQ，基础类库模式又分为两种使用方式：ACN（AN 和 CN 整合）和 JN 作为独立服务；CN、AN 和 JN 作为独立服务。由于两种模式，JN 使用方式一样，这里重点讲 ACN 整合与 CN、AN 分别独立服务的使用不同，JN 使用最后统一给出。

不管选用基础类库模式两种的哪一种，要使用框架的前提步骤是：

- 首先从 light_drtc 官方网址：GitHub 下载最新源码包，解压后，进入项目根目录。
- 其次使用命令：mvn package，打包成“light_drtc-${current_version}.jar”引入自己项目。

接下来将分两种方式分别介绍这两种模式的具体使用步骤。

2.5.1 ACN（AN 和 CN 整合）作为独立服务

选择使用此模式，意味着开发者在收集实时数据流时采用的 MQ 是 Kafka 或 RabbitMQ，框架中针对两者已经给出了具体实现，此时开发者不需要再关心 Kafka 或 RabbitMQ 如何收集数据，只需要把框架配置文件中的 MQ 相关属性切换成你自己的环境配置即可。

相比较 JN 所完成的实时计算任务而言，CN 和 AN 只是协调资源分配及任务统一分发，服务所在服务器压力相对较小，此时 CN 和 AN 两部分整合在一起，便于服务器资源的统一管理，何况作为计算型的服务器，当前配置普遍比较高，CN 和 AN 合在一起也是一种不错选择。此时从开发者使用框架角度而言，仅需要参考如下步骤即可。

1. 实现对所接受的MQ实时数据解析应用类

首先要实现框架中的“org.light.rtc.base.StreamLogParser.java”接口，源码如下：

```
public interface StreamLogParser {
    /**
     * 自定义实现：从指定队列 dataQu 中，获取指定数目 curNum 的单条日志信息，
     * 加工成每条信息类似：{uid:设备 ID 或通行证 ID，data:
     * {view:{docIds}，collect:{docIds}}}形式的信息列表
     * @paramdataQu
     * @paramcurNum
     * @return
     */
    List<String>parseLogs(ConcurrentLinkedQueue<String>dataQu,
int curNum);
}
```

实现相应业务逻辑，具体参考实例为测试包中的“org.light.ldrtc.parser.LogParser”，源码如下：

```
package org.light.ldrtc.parser;

import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.concurrent.ConcurrentLinkedQueue;

import org.light.rtc.base.StreamLogParser;

import com.alibaba.fastjson.JSONObject;
//日志解析测试用例，仅供参考，实际应用，需要开发者自行完成
//实现框架预定义的日志解析接口
public class LogParser implements StreamLogParser {

    /**
     * 测试：MQ 日志行，本测试，多个字段以“,”隔开，比如原始日志形式：
     * "deviceId,docId,type,timestamp",加工成每条信息类似：
     *: "{uid:设备 ID 或通行证 ID，data:{view: {docIds},
     * collect:{docIds}}}"形式的信息列表，例如：[{"uid":"light",
     * "data":{"view":["yl20160401", "yl20160402","yl20160403",
     * "yl20160404"]}},{"uid":"momo","data":{"view":
     * ["ty20160404","ty20160405","yl20160405","yl20160407"],
     * "share":["ty20160404"]}},{"uid":"taotao","data": {"view":
     * ["yl20160402",  "yl20160403","yl20160405"],
     *  "collect":["tu20160404"]}}]
```

```
         */
        @Override
        public List<String> parseLogs(
                ConcurrentLinkedQueue<String> dataQu, int curNum) {
            Map<String,Map<String,Set<String>>>rtMap =
                    new HashMap<String,Map<String,Set <String>>>();
            String line = null;
            String[] fields = null;
            Map<String,Set<String>> actions = null;
            Set<String>docIds = null;
            for(int i=0; i<curNum; i++){
                line = dataQu.poll();
                fields = line.split(",");//多个字段默认以“,”间隔
                if(fields!=null && fields.length==4){
                    actions = rtMap.remove(fields[0]);
                    if(actions==null){
                        actions = new HashMap<String,Set<String>>();
                    }
                    docIds = actions.remove(fields[2]);
                    if(docIds==null){
                        docIds = new HashSet<String>();
                    }
                    docIds.add(fields[1]);
                    actions.put(fields[2], docIds);
                    rtMap.put(fields[0],actions);
                }
            }
            List<String>rtList = new LinkedList<String>();
            JSONObjectrtJson = null;
            for(Entry<String,Map<String,Set<String>>> item
                                        : rtMap.entrySet()){
                rtJson = new JSONObject();
                rtJson.put("uid",item.getKey());
                rtJson.put("data",item.getValue());
                rtList.add(rtJson.toJSONString());
            }
            return rtList;
        }

        public void test(){
            ConcurrentLinkedQueue<String>dataQu          =          new
ConcurrentLinkedQueue<String>();
            dataQu.add("light,yl20160401,view,1234");
            dataQu.add("light,yl20160402,view,1234");
            dataQu.add("light,yl20160403,view,1234");
            dataQu.add("light,yl20160404,view,1234");

            dataQu.add("taotao,yl20160402,view,1234");
            dataQu.add("taotao,yl20160403,view,1234");
            dataQu.add("taotao,tu20160404,collect,1234");
            dataQu.add("taotao,yl20160405,view,1234");

            dataQu.add("momo,ty20160404,view,1234");
```

```
        dataQu.add("momo,ty20160404,share,1234");
        dataQu.add("momo,ty20160405,view,1234");
        dataQu.add("momo,yl20160405,view,1234");
        dataQu.add("momo,yl20160407,view,1234");

        int curNum = dataQu.size();

        this.parseLogs(dataQu, curNum);
    }
}
```

2. 启动AN节点服务进程

启动 AN 节点服务进程，一般根据需要至少启动 2 个 AN 服务，这里最好将多个 AN 放在不同服务器上，参考代码如下：

```
import org.light.ldrtc.parser.LogParser;
import org.light.rtc.admin.AdminNodeKafkaRun;
//import org.light.rtc.admin.AdminNodeRabbitMqRun;

public class AdminNodeServer {

    public void run(){
        //使用整合 RabbitMQ 的 AN
//      AdminNodeRabbitMqRun anr = new AdminNodeRabbitMqRun();

        //使用整合 Kafka 的 AN
        AdminNodeKafkaRunanr = new AdminNodeKafkaRun();

        //AN 和 CN 分别作为独立执行的服务进程
        //AdminNodeRunanr = new AdminNodeRun();
        //配置解析日志数据流主应用对象
        anr.setSteamParser(new LogParser());
        anr.run();
    }

    public static void main(String[] args) {
        AdminNodeServeradminServer = new AdminNodeServer();
        adminServer.run();
    }
}
```

此时即完成了 AN 节点启动的流程。此处需要注意的是，启动 AN 前，应该首先启动相应 JN，否则就会引发一系列错误。

2.5.2 CN、AN 作为独立服务

选用此种模式，意味着，服务器资源比较富裕，CN、AN 和 JN 分别作为独立服务进程存在，本节重点讲 CN 和 AN 两个部分的开发步骤。

1．数据流实时收集CN

实时收集数据一般用 MQ，目前比较活跃的开源 MQ 主要有 RabbitMQ 和 Kafka，CN 对所接受的每条数据默认存在本地队列缓存中，后台定时任务会将本地队列中收到的数据，按照每隔${mqDoBatchTimer}秒批量提交给 AN。框架中已经实现了针对 RabbitMQ 和 Kafka 的两个独立应用，如果选择其他 MQ 则需要用户自行实现。这其中首先需要配置好 MQ 实际环境。

(1) 如果读者 MQ 技术选型选用 Kafka 或 RabbitMq，可以直接使用框架中已实现的 2 个 MQ 应用：org.light.rtc.mq.KafkaMqCollect.java。

```
org.light.rtc.mq.RabbiMqCollect.java
```

直接实例化，调用相关方法即可。实际开发实例可以参考：

src/test/java/org/light/rtc/test/KafkaRabbitMqCollect.java，源码如下：

```
import org.light.rtc.mq.KafkaMqCollect;
import org.light.rtc.mq.RabbitMqCollect;

public class KafkaRabbitMqCollect {
    public void run(){
        //如果选用 Kafka
        KafkaMqCollectkmc = new KafkaMqCollect();
        kmc.collectMq();
        //如果选用 RabbitMq
        //  RabbitMqCollectrmc = new RabbitMqCollect();
        //  rmc.run();
    }

    public static void main(String[] args) {
        KafkaRabbitMqCollect krc = new KafkaRabbitMqCollect();
        krc.run();
    }

}
```

（2）如果读者选用其他 MQ，可以参考框架中 KafkaMqCollect.java 或 RabbiMqCollect.java 实现，应用需要继承 org.light.rtc.base.MqConsumer.java，对所接受的每条数据，代码中调用“this.mqTimer.parseMqText(userId, logText)”即可。

下面是使用 Kafka 实时收集数据流的完整代码，供选用其他 MQ，开发者自行实现时参考。

```
package org.light.rtc.mq;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

import org.light.rtc.base.MqConsumer;
import org.light.rtc.util.Constants;

import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
import kafka.message.MessageAndMetadata;
import kafka.serializer.StringDecoder;
import kafka.utils.VerifiableProperties;

public class KafkaMqCollect extends MqConsumer{//继承父类
    //定义消费者
    private ConsumerConnector consumer;

    public KafkaMqCollect(){
        super();//父类初始化
        this.init();
    }
    //kafka 基本属性配置
    public void init(){
        Properties props = new Properties();
        props.put("zookeeper.connect", Constants.kfZkServers);
        props.put("group.id", Constants.kfGroupId);
        props.put("auto.offset.reset",
                          Constants.kfAutoOffsetReset);
        props.put("zookeeper.session.timeout.ms", "4000");
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");
        props.put("serializer.class",
                         "kafka.serializer.StringEncoder");
        ConsumerConfig config = new ConsumerConfig(props);
```

```
            consumer = Consumer.createJavaConsumerConnector(config);
        }
        //实时收集数据
        public void collectMq(){
            Map<String, Integer>topicCountMap =
                        new HashMap<String, Integer>();
            topicCountMap.put(Constants.kfTopic, new Integer(1));

            StringDecoderkeyDecoder =
                    new StringDecoder(new VerifiableProperties());
            StringDecodervalueDecoder =
                    new StringDecoder(new VerifiableProperties());

            Map<String, List<KafkaStream<String, String>>>consumerMap
                        = consumer.createMessageStreams(
                            topicCountMap, keyDecoder,valueDecoder);

            KafkaStream<String, String> stream
                        = consumerMap.get (Constants.kfTopic).get(0);
            ConsumerIterator<String, String> it = stream.iterator();
            MessageAndMetadata<String, String>msgMeta;
            while (it.hasNext()){
                msgMeta = it.next();
                //调用父类方法,将含有相同主键的数据定位到固定 AN 对应本地队列中
                //以方便提交给相应 AN。
                super.mqTimer.parseMqText(msgMeta.key(),
                                            msgMeta.message());
                System.out.println(msgMeta.key()
                                    +"\t"+msgMeta.message());
            }
        }
    //  public static void main(String[] args) {
    //      KafkaMqCollectkmc = new KafkaMqCollect();
    //      kmc.collectMq();
    //  }
    }
```

2. 任务管理AN

对于所接受的来自 CN 的实时数据流的数据,AN 每隔${rtcPeriodSeconds}秒,将所收集的每条信息,统一加工成形如如下聚合 JSON 格式:

```
"{uid:设备 ID 或通行证 ID, data: {view:{docIds},collect:{docIds}}}"
```

的信息列表。具体开发可以参考“src/test/java/org/light/ldrtc/test/AdminNodeServer.java”。

日志解析随实际业务的不同有很大不同，因此这里需要开发者自行实现实时数据流的日志解析，需要实现“org.light.rtc.base.StreamLogParser.java”接口，具体可参考实例：

```
src/test/java/org/light/ldrtc/parser/LogParser.java
```

同样启动 AN 服务时，如上节一样，需要把日志解析类对象传递给框架。

上述 2.5.1 和 2.5.2 两节，只是运行方式不同，但本质上 CN 和 AN 都还是比较独立的模块，下面重点讲解 JN 开发步骤。

2.5.3 任务计算 JN

业务逻辑千变万化，这部分主要依靠开发者自定义实现，开发者需要实现框架定义的“org.light.rtc.base.**JobStatsWindow.java**”接口，源码如下：

```
public interface JobStatsWindow {
    /**
     * 根据接收的实时数据流，完成相关实时计算业务需求。
     * @paramuserLogs
     * @return
     */
    int rtcStats(List<String>userLogs);
    /**
     * 适合做线下离线批量处理计算
     * @param start
     * @param size
     * @return
     */
    int batchStats(int start, int size);
    /**
     * 返回服务健康状态
     * @return
     */
    int getHealthStatus();
}
```

这三个方法中，第 1 个和第 3 个是实时计算所必需的，当然第 2 个接口是针对间隔周期比较的分布式计算任务设计的，待后期扩展需要。

这里建议大家参考 JDK 本身提供的 Fork/Join 并行计算框架，以更高效率地利用服务器资源。开发实例入口参考“src/test/java/org/light/ldrtc/test/JobNodeServer.

java”，实例介绍的是一个将原始数据加工成框架所要求的聚合 json 数据形式，具体实现参考“org.light.ldrtc.jober.JobService.java 及 StatsTask.java”。

其中 JobService.java 源码如下：

```
package org.light.ldrtc.jober;

import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;

import org.light.rtc.base.JobStatsWindow;
import org.light.rtc.util.ConfigProperty;

public class JobService implements JobStatsWindow{

    //暂时为实现间隔周期比较长的批量计算接口
    @Override
    public int batchStats(int start, int size) {
        return 0;
    }

    //实时返回服务健康状况，实际应用应根据实际需要定义
    @Override
    public int getHealthStatus() {
        return 1;
    }

    //实时计算入口
    @Override
    public int rtcStats(List<String>userLogs) {
        System.out.println(ConfigProperty.getCurDateTime()
                    +" 计算之初分配的用户信息个数："+userLogs.size());
        //启用 fork/join 入口
        ForkJoinPoolfjPool = new ForkJoinPool();
        //调用实时计算业务实现类（继承 fork/join
        //带有返回值的父类 RecursiveTask)
        Future<Integer>fjTask
                = fjPool.submit(new StatsTask(userLogs));
        int rtNum = -1;
        try {
            rtNum = fjTask.get();
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        fjPool.shutdown();
        fjPool = null;
        userLogs.clear();
        userLogs = null;
```

```
        System.out.println(ConfigProperty.getCurDateTime()
                           + " 计算结束有效用户信息个数："+rtNum);
        return rtNum;
    }
}
```

StatsTask.java 源代码如下：

```
package org.light.ldrtc.jober;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.RecursiveTask;

import org.light.rtc.util.Constants;
import com.alibaba.fastjson.JSONObject;
//带有返回值的任务父类
public class StatsTaskextends RecursiveTask<Integer> {
    private static final long serialVersionUID = 1L;
    private List<String>uActions;

    /**
     * @paramuserLogs
     * userLogs 中每条数据都是事先加工好的类似："{'uid' : 设备 ID 或通行
     * 证 ID, 'data':{'view':{docIds},'collect':{docIds}}}"形式的
     *：信息列表，例如 userLogs 值为： [{"uid":"light","data":{"view":
     * ["yl20160401","yl20160402", "yl20160403","yl20160404"]}},
     *{"uid":"momo","data":{"view":["ty20160404","ty20160405",
     *"yl20160405","yl20160407"],"share":["ty20160404"]}}]
     */
    public StatsTask(List<String>userLogs){
        this.uActions = userLogs;
    }

    @Override
    public Integer compute(){
        int rtNum =-1;
        //如果当前任务数据集小于系统设定的单次计算任务数据最小值，
        //则单线程执行即可
        if(this.uActions.size()<Constants.minJobBatchNum)
            Map<String,List<String>>tmpActions = null;
            JSONObjectrtJson = null;
            //待加工最终数据形态
            Map<String,Map<String,List<String>>>uActions
                = new HashMap<String,Map<String,List <String>>>();
            for(String rt:this.uActions){
                rtJson = JSONObject.parseObject(rt);
                tmpActions = (Map<String,List<String>>)rtJson
                                    .get ("data");
```

```
                uActions.put(rtJson.getString("uid"),
                                                    tmpActions);
            }
            System.out.println("加工后待计算的初始数据："
                                                +uActions. size());
            //这里就可以实现自己业务逻辑了，这里实例只是展示加工好的
            //数据而已
            for(Entry<String,Map<String,List<String>>> item
                                        : uActions.entrySet()){
                System.out.println(item);
            }
            rtNum = uActions.size();
        }else{
        //开始任务拆分，递归调用，直至拆分成可以独立快速计算的小任务为止
            int middle = this.uActions.size()/2;
            StatsTask left = new StatsTask(
                            this.uActions.subList(0, middle));
            StatsTask right = new StatsTask(this.uActions.
                      subList (middle, this.uActions.size()));
            left.fork();
            int leftNum = left.join();
            int rightNum = right.compute();
            //最终结果合并
            rtNum = leftNum + rightNum;
        }
        return rtNum;
    }
}
```

上述 JobService.java 及 StatsTask.java 实例展示了框架实现实际业务计算的基本流程。当然这里只是笔者所建议的方式，只要类似 JobService.java 实现框架计算接口，就可以选用其他方式实现具体业务逻辑，只是目前看 JDK 单进程内，fork/join 所实现的多线程计算优势更大。

2.6 总结

本章从框架简介、框架代码组织结构、与 Storm 的对比，以及详细的使用说明方面，让读者朋友对 light_drtc 框架初步有个直观印象，下章将会给大家介绍框架设计原理思想及核心代码实现。

第 3 章
light_drtc 核心技术实现

前一章主要对 light_drtc 框架进行初步介绍，让读者朋友先有个直观印象，本章接续前章将重点对框架设计原理、技术架构及核心模块实现方面进行介绍，希望读者对 light_drtc 有更深刻的认识。

3.1 light_drtc 技术架构

本节主要向读者介绍作者自研框架 light_drtc 的技术架构，包括逻辑架构和物理架构，并向读者介绍其详细设计流程。light_drtc 主要参考当下流行的批量计算框架 Hadoop 的 Map/Reduce、实时计算框架 Spark Streaming 的 Mini-batch 和 Storm 的 Spout/Bolt 处理思想，技术实现上主要使用 **MQ（Kafka/Rabbit）实时接收数据，RPC 服务（Thrift 封装）调用**（CN 每隔 N 秒统一调用 AN 的 RPC 服务向其推送数据，AN 每隔 *K* 秒将当前所接受的数据加工成类 Spark 的 RDD 形式，然后统一调用 JN 的 RPC 服务，将计算分发并协调资源优化配置）、**Java7 开始具备的 Fork/Join 计算框架**。

图 3.1 所示为作者本人自研的一套分布式实时计算框架 light_drtc，整个计算框架从逻辑上主要分为三部分：数据实时收集服务模块 CollectNode（CN）、任务

协调管理模块 AdminNode（AN）和任务计算模块 JobNode（JN）。

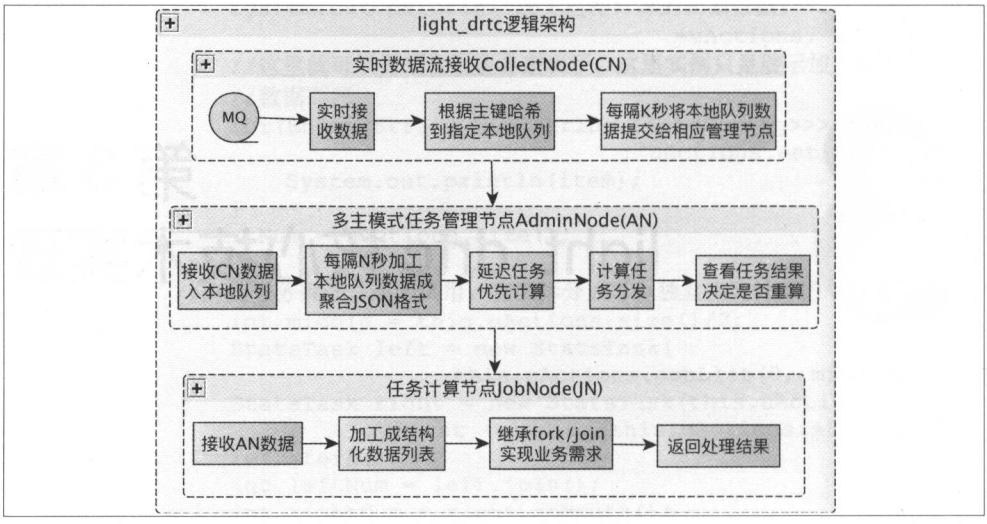


图 3.1 light_drtc 逻辑架构

该框架运行流程如下：

（1）数据收集 CollectNode（CN）：终端用户的实时点击行为通过 MQ 统一收集，将数据暂存在本地双端队列中，每隔 *N* 秒批量发送到计算集群中的任务管理节点。

（2）任务协调管理 AdminNode（AN）：对于数据收集节点所发送过来的数据，首先保存在本地双端队列中，后台定时任务每隔 *K* 秒获取当前队列中的内容，然后统一分发给任务计算节点，并自动协调相关资源分配，确保任务计算节点中不存在两个任务同时并行的问题，同时保证批计算任务的顺序进行。

（3）任务计算服务 AdminNode（AN）：对所接收的批量用户行为数据，根据实际业务需求，利用 JDK7 开始具备的 Fork/Join 编程框架，进行相应的处理。

整个架构虽然从逻辑上可以分为 CN、AN 和 JN 三个相互独立的模块，但考虑到系统维护成本，在物理架构上，系统略做了调整，图 3.2 所示为笔者自研框架 light_drtc 物理架构示意图，从这个图中，可以看出框架的层级结构。

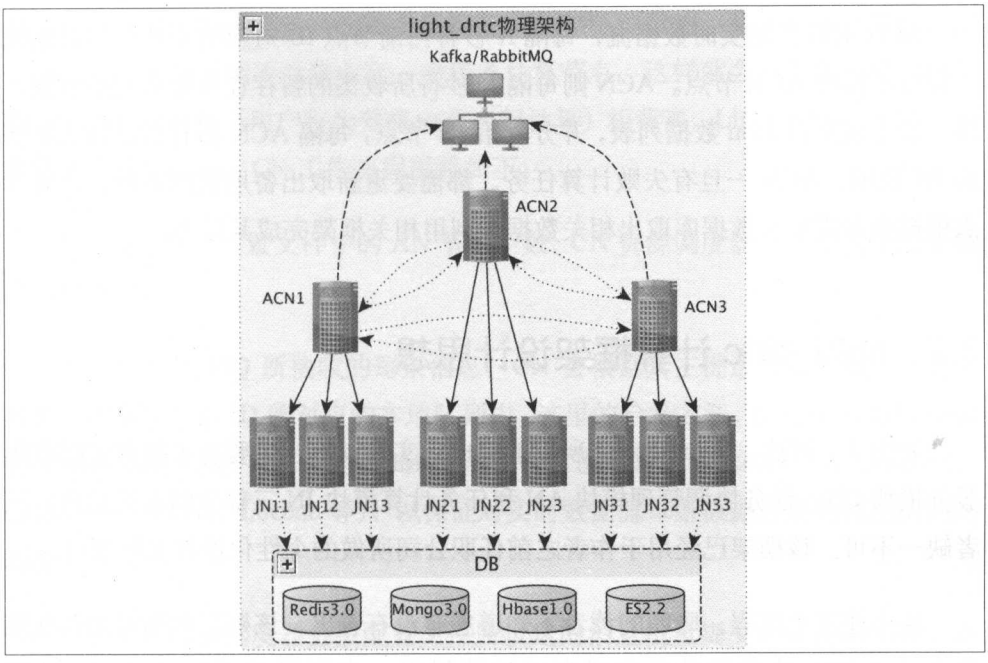


图 3.2 light_drtc 物理架构

框架 light_drtc 源头来自用户的实时行为 MQ 消息，框架目前已经集成当前 2 个比较热门的开源 MQ：Kafka 和 RabbitMQ，框架中 ACN 所在服务器节点是 MQ 的消费端，这其中，ACN 本身囊括了框架 light_drtc 中的 CN 和 AN 两部分功能，因此此处命名为 ACN 服务节点。为了保证计算框架的高可用，集群中 ACN 一般设置 3 个节点，每个 ACN 节点均可独立管理自己的 JN 节点。从整个架构中可以看出，每个 ACN 及其所管辖的 JN 节点集群可以作为框架计算中逻辑及物理上的一个独立计算单元，计算单元之间没有任何关联，高度独立，一个计算单元的故障，不会影响整个计算集群。

进一步说，为了保证每个计算单元内的 HA，每个 ACN 一般根据实际情况，至少会独立负责 3 个 JN 节点，这样即使每个计算单元内有单个 JN 节点故障，也不会影响该计算单元的计算能力，这样会进一步保障整个计算集群的 HA。计算的最终结果一般都是存在 DB，因此，整个框架中最下层是 DB 层，这里笔者列出了当前易用且广为使用的几个 DB：Redis3.0、Mongo3.0、HBase1.0 和 ES2.2，根据业务实际需要，读者可以自行选择。

ACN 对收到的实时数据流，每隔 N 秒将当前节点 ID 对应的本地队列数据统一提交给相应 ACN 节点。ACN 则每隔 K 秒将所收集的暂存在本地队列的数据，统一加工成聚合 Json 数据列表，并分发给 JN 节点，每隔 ACN 都有自己独立管辖的 JN 范围。ACN 一旦有失败计算任务，都需要重新取出备用数据补救；计算节点根据业务需要从数据库取出相关数据，利用相关框架完成其任务。

3.2 light_drtc 计算框架设计思想

如图 3.1 所述，light_drtc 之所以能完成近实时计算，主要在于数据实时收集服务模块 CN、任务协调管理模块 AN 和任务计算模块 JN 三部分的有机整合，三者缺一不可。该框架已经用于作者之前任职公司所做的个性化推荐实际项目。

从上述三个流程，聪明的读者不知道您有没有看出，该框架和现有知名框架有什么不同？该框架的分布式实时计算思想，主要得益于目前 Hadoop 的 Map/Reduce、Spark Streaming 的 Mini-batch 和 Storm 的 Spout/Bolt 处理方式。下面将分别介绍 CN、AN 和 JN 三部分的设计思想，以方便读者更好地理解 light_drtc。

3.2.1 CN 设计思想

在面向实时数据流时，实时计算有两种方式：小批量计算和单条计算。且不论每个计算过程的复杂性，单单就高并发期间单条数据的实时计算任务的密集性，就很容易导致一些复杂计算的任务堆积（即使有后台任务调度），从而引发一系列未知系统异常。所以，这里作者借鉴 Spark Streaming 的 Mini-batch 方式处理实时数据流，即每隔 K 秒批量提交一下前 K 秒所接受的数据给任务管理节点，并同时存到 NoSQL，以备任务计算失败时重新计算。

大中型 APP 及网站日均 PV 非常高，高峰期往往每秒就有上万、或几十万甚至上百万 PV，应对此种情况，实际生产环境下，一般都采用多个 MQ 消费端，每个 MQ 消费端接收一部分用户行为数据，并提交给 AN。在 N（如果 N>5）秒时间内，往往一个用户会产生多个用户行为信息，由于 MQ 本身的负载，默认情

况下，一个用户的多条行为往往会分散在多个 MQ 消费端，相关任务计算时，单个用户的行为就可能不会集中在一个任务计算节点。这样就会与作者计算设计原则（单位时间内单个用户行为要集中在一起计算）相背离。Light_drtc 框架则很好地解决了这个问题。CN 工作流程思路如下：

（1）根据系统配置文件中的 AN 节点个数，CN 资源调度器会初始化相应个数的本地队列。

（2）对于从 MQ 所接收的每条信息，CN 会根据其主键进行哈希自动映射到其中一个 AN 节点 ID 所对应的本地队列中。这里的哈希方法，作者采用的 Google 的 Guava 所提供的 128 位的 MurmurHash（它也同时用于 Redis，Memcached，Cassandra，HBase，Lucene 等），以保证对实时数据流每条数据分发到相应队列的均衡。

（3）CN 内部调度器会按配置文件所设定的消息批量提交时间间隔周期，即 $mqDoBatchTimer 秒，每次提交数据给 AN 时，首先获取当前服务状态正常的 AN 节点列表，然后从内部相应各个本地队列中取出当前既定个数的信息，以 AN 节点个数相同的线程数将其分别提交给相应 AN 节点，多线程操作更是为了保证传递效率。

（4）如果有个别 AN 节点服务异常，那么，原本发往该失效 AN 的数据块，则会随机发往其他服务正常的 AN，这样保证最大程度的不丢失实时数据块。

上述流程即可保证实时数据流中的每条共享主键值的数据都会发往同一个 AN。这里需要注意的是，CN 传递给 AN 的仍然是所接受的原始日志，至于日志解析成框架所要求的聚合 JSON 形式的列表，则需要 AN 实现。

3.2.2 AN 多主模式设计思想

对于集群计算任务协调管理，这里计算任务的分发我们重点参考了 Storm 的 Spout/Bolt 处理方式，AN 相当于 Spout，JN 相当于 Bolt；对计算任务的资源协调管理。我们参考 Hadoop2.x 的 NameNode 的 Federation 和 Yarn 资源管理框架，对

AN 采用多主模式，每个 AN 都有自己独立的 JN 范围，彼此间没有交集，且计算任务都是独立的。计算任务的协调管理，旨在将上述 Mini-batch 的实时数据流统一加工成任务计算节点所需要的数据格式，并按照每批数据规模大小，自行选择所需计算任务节点个数。这里为保证计算任务的有序进行，我们对计算任务本身也做了任务排队，当上一个任务没有完成时，下个任务无法执行。AN 具体实现流程如下：

（1）AN 初始化，会分别设 3 个内置成员：

- 原始日志数据队列 sourceQu，用于接收从 CN 所传递的实时数据流。
- 内存中待执行任务数据集队列 memTaskQu，用于保留排队待执行的加工好的固定时间间隔的数据集。
- 内存中有序待执行任务 ID 集合 delayTaskTreeMap，用于保留超过内存中所设定的最大延迟个数$maxDelayTaskNum 的任务数据块，而以单个文件形式存在于指定延迟任务目录下的文件数据集 ID。

（2）AN 将从 CN 所收到的信息直接存入本地队列 sourceQu，然后按配置文件所设定的任务计算周期，即$rtcPeriodSeconds 秒，每次将 sourceQu 中当前时刻用户信息取出，统一加工成每条信息以用户设备 ID 或其他唯一标识为主键聚合的 JSON 列表 currentTaskData，数据格式形如"{uid:设备 ID 或通行证 ID，data:{view:{docIds},collect:{docIds}}}"。

（3）如果当前时刻，上批计算任务还没有结束，那么此时 AN 任务调度器要做的事情如下：

- 如果当前 memTaskQu 元素个数小于$maxDelayTaskNum，则直接将当前加工好的聚合 Json 列表 currentTaskData 存入 memTaskQu；
- 否则，取当前时间戳为文件名将 currentTaskData 存入延迟任务目录下，并记录在 delayTaskTreeMap 中。

（4）如果当前时刻，上批计算任务已经结束，那么此时 AN 任务调度器要做的事情如下：

- 如果当前 memTaskQu 元素个数为 0，则意味着无延迟任务，此时，可以直

接启动新线程将 currentTaskData 分发给 JN 计算。

- 如果当前 memTaskQu 元素个数大于 0，则有延迟任务，那么此时：
  - ➢ 首先从 memTaskQu 中取出延迟最久的数据块执行计算任务；
  - ➢ 如果当前 memTaskQu 元素个数大于$maxDelayTaskNum，此时将最新得到的 currentTaskData 写入延迟任务目录文件中；
  - ➢ 如果当前 memTaskQu 元素个数小于$maxDelayTaskNum，此时，首先需要判定 delayTaskTreeMap 是否存在延迟任务文件记录，如果有延迟任务文件记录，则直接逐一获取其中延迟最久的文件对应数据存入 memTaskQu，并同时删除该文件（避免延迟任务文件堆积），直到 memTaskQu 中待执行任务数据块个数达到$maxDelayTaskNum 或 delayTaskTreeMap 中延迟任务文件 ID 取完，因此此时还需要进行进一步判定：
    - ◆ 如果此时当前 memTaskQu 元素个数小于$maxDelayTaskNum，则直接将当前最新 currentTaskData 存入 memTaskQu。
    - ◆ 如果此时当前 memTaskQu 元素个数大于$maxDelayTaskNum，则获取当前时间戳，作为延迟任务目录下的最新文件名，将当前最新聚合数据块 currentTaskData 写入文件，并同时在 delayTaskTreeMap 记录。

（5）AN 任务调度器开始进行计算任务分发。

- 如果待执行数据块包含的聚合 Json 数据个数小于之前设定的任务计算最小阈值，则从当前服务正常的 JN 中随机挑选一个开始计算；
- 如果待执行数据块包含的聚合 Json 数据个数大于之前设定的任务计算最小阈值，根据当前 AN 所管辖的服务正常状态的 JN 个数，同时启动相应线程数，将任务均分。
- 如果期间计算任务失败，则需要二次重新计算。

经过上述流程，AN 作为计算任务调度器，可以很好地协调管理实时数据流的计算。

3.2.3 JN 设计思想

JN 即集群计算任务节点，就是最终的计算任务落脚点，根据具体业务需求，设计合适的计算模型，充分利用现在服务器多核、多线程优势，提高计算节点服务器资源利用率。由于该框架选择 Java 作为开发语言，这里为了保证单个计算任务的高效性，主要使用 JDK7 开始具备的 Fork/Join 框架，该框架设计本身类似于 Hadoop 的 Map/Reduce，更能充分利用计算机内核，保证计算任务执行的高效。

由于此部分和具体业务需求相关，因此，light_drtc 框架对这块给予了高度灵活性，框架只是定义了计算基本接口和框架运行基础架构，开发者只需要实现相关接口，具体使用参考前面的对 light_drtc 使用方法的描述。这里我们重点给大家介绍一下 JDK7 新增的 Fork/Join 模块。

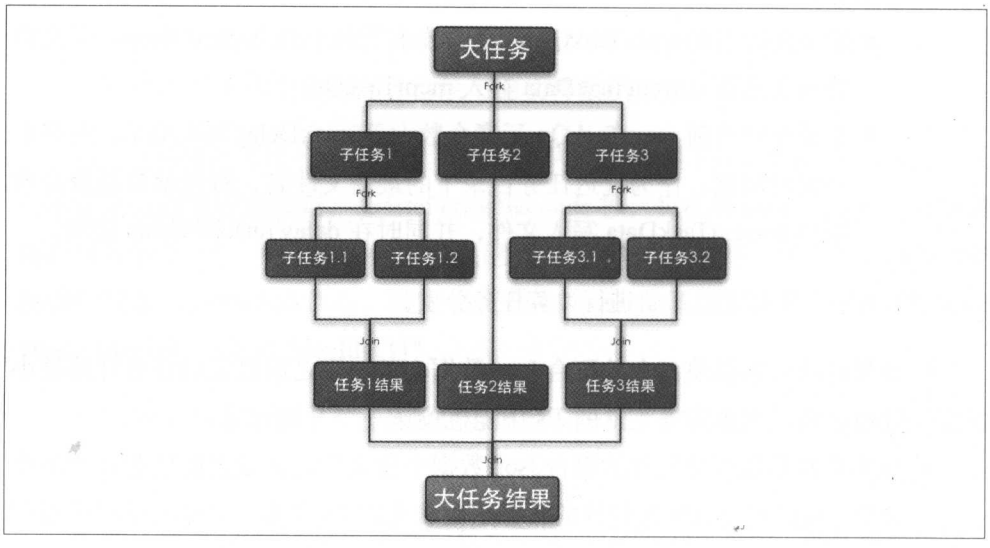


图 3.3 Fork/Join 框架工作流程

Fork/Join 模块是 Java7 提供的一个用于并行执行任务的框架[1]，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。Fork 就是把一个大任务切分为若干子任务并行的执行，Join 就是合并这些子任务的执

1 聊聊并发 Fork/Join 框架介绍 http://www.importnew.com/17565.html

行结果，最后得到这个大任务的结果。比如计算 1+2+…+10000，可以分割成 10 个子任务，每个子任务分别对 1000 个数进行求和，最终汇总这 10 个子任务的结果。Fork/Join 的运行流程图如图 3.3 所示。

Fork/Join 模块是对原来 Executors 的基础上增加了一种并行分治计算中的工作窃取（work-stealing）策略，即某个线程从其他队列里窃取任务来执行。这是和 Executors 这个方式最大的区别，更加有效地使用了线程的资源和功能。

那么为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，就可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如 A 线程负责处理 A 队列里的任务。但是有的线程会先把自己队列里的任务完成，而其他线程对应的队列里还有任务等待处理。

干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部取任务执行，而窃取任务的线程永远从双端队列的尾部取任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

Fork/Join 使用两个类来完成以上两件事情：

- **ForkJoinTask**：我们要使用 ForkJoin 框架，必须首先创建一个 ForkJoin 任务。它提供在任务中执行 fork()和 join()操作的机制，通常情况下我们不需要直接继承 ForkJoinTask 类，而只需要继承它的子类，Fork/Join 框架提供了以下两个子类：
  - ➢ RecursiveAction：用于没有返回结果的任务。
  - ➢ RecursiveTask：用于有返回结果的任务。

- **ForkJoinPool**：ForkJoinTask 需要通过 ForkJoinPool 来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

3.3 light_drtc 核心技术的实现

上节给读者详细介绍了 light_drtc 框架的设计思想，本节主要给读者介绍三大核心模块的主要技术实现，框架底层通信采用广为人知的 RPC。light_drtc 所涉及技术主要有：MQ 采用 RabbitMQ3.5.6 / Kafka0.9、RPC 服务采用 Thrift0.90.3，数据存储采用 Mongo3.0.7、Redis3.0.7 和 ES2.2。框架以 Java 为主开发语言，如果想使用其他语言开发，可以自行使用 Thrift 生成相应语言下的客户端 API，进而实现相关业务逻辑。

框架底层通信核心 API 如下述 Thrift 的 IDL 描述：

```
service TDssService{
    //CN 实时收集数据提交给 AN
    i32 addMqinfoToAdminQu(1:list<string>uLogs);
    //AN 调用 JN 完成实时计算
    i32 statsUsersAction(1:list<string>userLogs);
    //AN 调用 JN 完成批量计算
    i32 batchUsersAction(1:i32 start, 2:i32 size);
    i32 getAdminNodeId();  //获取 AN 节点 ID
    i32 getHealthStatus();   //获取节点健康状态
    i32 getJobStatus();      //获取当前节点的工作状态
}
```

整个框架围绕着这几个底层 API 将框架的三个核心模块：CN、AN 和 JN 连成一体。接下来将给大家介绍各个核心模块的技术实现。

3.3.1 实时收集数据 CN

实时数据流的收集，一般做法是：实时读取访问日志或通过前端埋点实时收集行为数据。这两种方式最终都会异步提交所收集的数据给 MQ。目前业内广为

使用的 MQ 主要有两个：RabbitMQ 和 Kafka，二者均为目前比较成熟且使用广泛的 MQ。后续章节会专门介绍 Kafka 和 RabbitMQ 的异同及使用方法。

如上节“3.2.1 CN 设计思想”所述，此部分，框架中并没有特别要求开发者选用固定的 MQ 选型，此处给开发者保留了高度灵活性，开发者可以根据自己业务需要，选择使用相应 MQ 接收数据。对于接收的每条数据，这里要求**每条数据都有一个类似于 userId（用户 ID）或 docId（文档 ID）的主键 ID，以便后续将含有相同主键的信息均集中在一个 AN 区域内做相应计算**。对于每条 MQ 信息，都将调用“2.5.2 CN、AN 作为独立服务”一节中的**开发者自实现的接口**“this.mqTimer.parseMqText(userId, logText)”，将其推送到 CN 本地资源调度器中的本地队列。此模块关键是 CN 资源调度器，这里将从代码角度向大家加以说明设计思想。

CN 资源调度器完整代码实例 MqDoTimer.java：

```
package org.light.rtc.timer;

import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.TimerTask;
import java.util.Map.Entry;

import org.light.rtc.client.LrtdcClient;
import org.light.rtc.util.ConfigProperty;
import org.light.rtc.util.Constants;
import org.light.rtc.util.Murmurs;

public class MqDoTimer extends TimerTask {
    //实时接收 MQ 信息的本地队列
    protected List<ConcurrentLinkedQueue<String>>dataStreamList
                = new ArrayList<Concurrent LinkedQueue<String>>();
    //AN 节点相关配置
    private String[] adminNodeHosts
                    = Constants.adminNodeHosts. split(",");
    private int adminNodeNum;
    private                 ConcurrentHashMap<Integer,LrtdcClient>
adminNodeClientMap;
    //Google 开源 Guava 中的 MurmurHash
    private Murmurs mmHash = new Murmurs();
```

```
private long timerNum = 0;

public MqDoTimer(){
    if(adminNodeHosts!=null){
        adminNodeNum = adminNodeHosts.length;
        adminNodeClientMap
            = new ConcurrentHashMap<Integer, LrtdcClient>();
        for(int i=0; i<adminNodeNum; i++){
            String[] tmpHostIp = adminNodeHosts[i].split(":");
            adminNodeClientMap.put(i,
                                   new LrtdcClient(tmpHostIp[0],
                                Integer.parseInt (tmpHostIp[1])));
            dataStreamList.add(
                        new ConcurrentLinkedQueue <String>());
        }
    }
}

//MQ 所收集的每条数据均通过 MurmurHash 添加到相应 AN 节点 ID 所对应队列中
public void parseMqText(String routKey, String mqText){
    int adminId = (int)
          (mmHash.getGuavaMurmurs(routKey) %adminNodeNum);
    this.dataStreamList.get(adminId).add(mqText);
}

//获取当前有效 AN
public List<Integer>getAbleAdminNodeIds(){
    List<Integer>ableJobNodeIds = new ArrayList<Integer>();
    for(Entry<Integer,LrtdcClient> item
                        : adminNodeClientMap. entrySet()){
        if(item.getValue().getHealthStatus()==1){
            ableJobNodeIds.add(item.getKey());
        }
    }
    return ableJobNodeIds;
}

//CN 资源调度器核心实现
@Override
public void run() {
    try{
        int tmpNum = -1;
        //获取当前每个 AN 节点 ID 所对应本地队列的当前信息个数
        Map<Integer,Integer>dataQuNums
                        = new HashMap<Integer, Integer>();
        for(int i=0; i<dataStreamList.size(); i++){
            tmpNum = dataStreamList.get(i).size();
            if(tmpNum>0){
                dataQuNums.put(i, tmpNum);
            }
        }
        //AN 节点所对应的本地队列中只要有 1 个含有实时数据
```

```
        if(dataQuNums.size()>0){
            System.out.println(ConfigProperty.
                    getCurDateTime()+" currently will be doing
                    adminNode and logNum : "+dataQuNums);
            //获取当前有效 AN
            List<Integer>enableANIds
                                = this.getAbleAdminNode Ids();
            if(enableANIds.size()>0){
                for(Entry<Integer,Integer>dqnItem
                                : dataQuNums.entrySet()){
                //启动新线程将当前 AN 节点 ID 所对应队列中数据提交给 AN
                    new Thread(new BatchDoLog(dqnItem.getKey(),
                                    dqnItem.getValue(),
                                enableANIds)).start();
                }
            }else{
                System.out.println(ConfigProperty.
                    getCurDateTime()+" currently there is no
                    enable adminNode for receving stream logs");
            }
        }else{
            System.out.println(ConfigProperty.
                        getCurDateTime()+"\t 当前时刻无消息 ");
        }
    } catch (Exception e) {
        System.err.println("用户行为日志收集错误"
                                    +e.getMessage());
    }
    //每处理 1000 批数据，进行垃圾回收
    if(++timerNum%1000==0){
        System.out.println("实时数据流按固定周期提交 "
                                        +timerNum+" 次");
        System.gc();
    }
}

//内部类：完成 Mini-Batch 信息提交给 AN
protected class BatchDoLog implements Runnable{
    private int adminId;//AN 节点 ID
    private int logNum;//当前日志队列中信息个数
    private List<Integer>enableAnIds;//当前可用 AN 节点 ID 数组

    public BatchDoLog(int adminId, int logNum,
                        List<Integer> enableAdminIds){
        this.adminId = adminId;
        this.logNum = logNum;
        this.enableAnIds = enableAdminIds;
    }

    private Random rand = new Random();
```

```
        @Override
        public void run() {
            List<String>tmpLogs = new LinkedList<String>();
            ConcurrentLinkedQueue<String>tmpQu
                                = dataStreamList. get(adminId);
            for(int i=0;i<logNum;i++){
                tmpLogs.add(tmpQu.poll());
            }
            if(!enableAnIds.contains(adminId)){
                adminId = enableAnIds.get(
                            rand.nextInt(enableAnIds.size()));
            }
            //CN提交当前批次实时数据给AN，即调用框架底层的API实现
            adminNodeClientMap.get(adminId)
                                .addMqinfoToAdminQu(tmpLogs);
        }
    }

}
```

注意：代码实例中，关键处均加了注释，再结合前述设计思想，相信读者应该可以看懂，当然您也可以和作者随时交流，以下代码处均同上说明。

3.3.2 任务协调管理 AN

该模块连接实时收集数据模块和任务计算模块，对于数据收集节点所定时推送过来的数据，首先保存在本地队列（借助 JDK 内核的并发编程包中的 ConcurrentLinkedQueue 实现）中，然后任务管理节点每隔 K 秒将当前队列中数据加工成任务计算所需格式，这里为了兼容各种业务数据需要，统一采用了 JSON 字符串格式，根据事先评估的计算任务节点的最大执行能力，计算出每批数据需要几个计算任务节点执行，从任务计算集群中挑选出空闲节点，数据均分，同时保证每批计算任务的顺序执行，上一个计算任务没有结束，下个计算任务等待。

如图 3.1 所示，light_drtc 框架中，设置多主模式的任务管理节点 AN 集群，每个 AN 节点均独自管辖下属所有 JN 节点，此模块在开发上仅需要开发者完成 MQ 实时日志数据的解析接口。日志解析随实际业务有很大不同，因此这里需要开发者自行实现实时数据流的日志解析，需要实现接口“org.light.rtc.base.StreamLogParser.java”，可以参考实例：“src/test/java/org/light/ldrtc/parser/LogParser.java”。

AN 也是 light_drtc 框架的核心模块，本节主要向读者介绍任务调度和任务分

发两部分，结合“3.2.2 AN 多主模式设计思想”，请读者朋友仔细看如下代码实例。

AN 任务调度器完整代码实例 AdminNodeTimer.java

```
import java.util.List;
import java.util.TimerTask;
import java.util.TreeMap;
import java.util.concurrent.ConcurrentLinkedQueue;
import org.light.rtc.admin.AdminNodeService;
import org.light.rtc.base.StreamLogParser;
import org.light.rtc.util.ConfigProperty;
import org.light.rtc.util.Constants;
import org.light.rtc.util.FileUtil;

public class AdminNodeTimer extends TimerTask{
    //JDK8 的文件读写工具类
    protected FileUtil fu = new FileUtil();
    //接收来自 CN 的原始实时数据流队列
    protected static ConcurrentLinkedQueue<String>dataQu
                        = new ConcurrentLinkedQueue <String>();
    //计算任务延迟计算文件目录下文件所对应的文件名（时间戳）及数据个数
    private static TreeMap<Long,Integer>delayTaskIdDataNums
                        = new TreeMap<Long,Integer>();
    //优先计算的内存级延迟任务对应数据块队列（计算加工好的 JSON 格式）
    protected     static     ConcurrentLinkedQueue<List<String>>
memDelayQu = new ConcurrentLinkedQueue <List<String>>();
    //当前时刻的上批小批量计算任务状态：0，空闲； 1，忙碌
    private static int minBathJobStatus = 0;
    //框架定义的日志解析接口，需要开发者传入相应实现类
    private static StreamLogParser slp;
    //AN 中的任务分发器
    private AdminNodeService ans = new AdminNodeService();
    private long timerNum = 0;

    //开发者自定义实现的日志解析类
    public static void setStreamLogParser(
                        StreamLog ParserstreamLogParser){
        slp = streamLogParser;
    }

    //接收 CN 所传递过来的原始日志数据
    public static boolean addSteamData(List<String>uLogs){
        return dataQu.addAll(uLogs);
    }

    //当前任务状态
    public static int getJobStatus(){
        return minBathJobStatus;
    }
```

```
        //任务调度器核心实现模块
        @Override
        public void run() {
            //获取当前原始日志数据大小
            int curLogNum = dataQu.size();
            System.out.println(ConfigProperty.getCurDateTime()+"
curLogNum : "+curLogNum);
            if(curLogNum>0){//只有大于0时才执行计算
                //用户自定义实现解析后的json格式数据列表
                List<String>userActionList=     slp.parseLogs(dataQu,
curLogNum);
                //当前内存级延迟任务数量
                int memDelayTaskNum = memDelayQu.size();
                if(minBathJobStatus==0){//当前任务状态空闲时
                    if(memDelayTaskNum>0){//内存里有延迟任务优先计算延迟
任务
                        //为当前即将计算的任务分配延迟最久的数据集
                        ans.setUserActions(memDelayQu.poll());
                        //启动后台线程执行小批量任务计算
                        new Thread(new AdminConsole()).start();
                        //判定当前内存级延迟任务数量是否达到上限
                        if(memDelayTaskNum
                                <Constants. maxDelayTaskNum){
                            //如果延迟任务文件个数大于0
                            if(delayTaskIdDataNums.size()>0){
                               //将延迟最久的任务文件获取数据塞入内存
                               //及延迟任务数据集
                                while(memDelayQu.size()
                                        <Constants. maxDelayTaskNum
                                    && delayTaskIdDataNums. size()>0){
                                    //延迟任务文件目录
                                    String taskFile
                                            = Constants. delayTaskDir
                                + delayTaskIdDataNums. pollFirstEntry()
                                  .getKey() //文件名
                                  //延迟任务文件后缀名
                                + Constants.delayTaskFileSurfix;
                                    //读取延迟任务文件对应数据
                                    List<String>rtJsonList
                                        = fu.readActions(taskFile);
                                    if(rtJsonList!=null){
                                        memDelayQu.add(rtJsonList);
                                    }
                                    //删除已读取的延迟任务文件
                                    fu.delActionFile(taskFile);
                                }
                            }
                            //此时判定内存级延迟任务数是否达到上限
                            if(memDelayQu.size()
                                    <Constants. maxDelayTaskNum){
```

```
                    //将当前最新加工的行为数据塞入内存级延迟任务集
                        memDelayQu.add(userActionList);
                    }else{
                        long rtNum = ans.getCurrentTime();
                    //将当前最新加工行为数据集写入延迟任务文件目录
                        delayTaskIdDataNums.put(rtNum,
                                                curLogNum);
                        fu.writeActions(userActionList,
                            Constants.delayTaskDir + rtNum
                            + Constants.delayTaskFileSurfix);
                    }
                }else{
                    //当前内存及延迟任务已到达上限,
                    //直接将最新数据集写入文件
                    long rtNum = ans.getCurrentTime();
                    delayTaskIdDataNums.put(rtNum,
                                                curLogNum);
                    fu.writeActions(userActionList,
                            Constants.delayTaskDir+rtNum
                            + Constants.delayTaskFileSurfix);
                }
            }else{
                ans.setUserActions(userActionList);
                new Thread(new AdminConsole()).start();
            }
        }else{
            //当前工作状态繁忙时，将当前加工好的数据集处理
            //根据内存集延迟任务数量是否达到上限
            if(memDelayTaskNum<Constants.maxDelayTaskNum){
                //塞入内存
                memDelayQu.add(userActionList);
            }else{//写入文件
                long rtNum = ans.getCurrentTime();
                delayTaskIdDataNums.put(rtNum, curLogNum);
                fu.writeActions(userActionList,
                        Constants.delayTaskDir+rtNum
                        +Constants. delayTaskFileSurfix);
            }
            System.out.println(ConfigProperty
                    .getCurDateTime()+" 上次任务尚没结束");
        }
    }
    //每 100 次任务批量处理进行一次垃圾回收
    if(++timerNum%100==0){
        System.out.println("分布式计算任务周期频率更新 "
                                            +timerNum);
        System.gc();
    }
}

//当内存延迟任务存在时，调度器会一直循环获取延迟任务数据集计算，
```

```
    //直到没有延迟任务
    public void reRun(){
        if(memDelayQu.size()>0){
            System.out.println("AdminNodeTimer reRun method ...."
                                +memDelayQu.size());
            //获取内存延迟任务数集
            ans.setUserActions(memDelayQu.poll());
            //开始计算延迟最久数据集任务
            new Thread(new AdminConsole()).start();
            //延迟任务文件存在时
            if(delayTaskIdDataNums.size()>0){
                String taskFile = Constants.delayTaskDir
                + delayTaskIdDataNums.pollFirstEntry().getKey()
                + Constants.delayTaskFileSurfix;
                //获取延迟任务文件数据
                List<String>rtJsonList
                                = fu.readActions (taskFile);
                if(rtJsonList!=null){
                    memDelayQu.add(rtJsonList);
                }
                fu.delActionFile(taskFile);//删除延迟任务文件
            }
        }
    }
    //内部类实现小批量任务计算
    protected class AdminConsole implements Runnable{

        @Override
        public void run() {
            minBathJobStatus = 1;
            try{
                ans.run();
            } catch (Exception e) {
                System.err.println("分布式计算任务管理服务错误： "
                                        +e.getMessage());
            }
            minBathJobStatus = 0;
            //判定内存延迟任务是否存在，若存在则一直不停的循环计算
            reRun();
        }
    }
}
```

AN任务分发器完整代码实例 AdminNodeTimer.java

```
package org.light.rtc.admin;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.Util.Date;
import java.util.HashMap;
```

```
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import org.light.rtc.client.LrtdcClient;
import org.light.rtc.util.Constants;
import org.light.rtc.util.ConfigProperty;

public class AdminNodeService {
    //JN 配置：每个 AN 都有自己的 JN 范围
    private String[] jobNodeHosts
                    = Constants.jobNodeHosts.split(",");
    private int jobNodeNum = jobNodeHosts.length;
    private                 ConcurrentHashMap<Integer,LrtdcClient>
jobNodeClientMap;
    //已经加工好的 JSON 格式数据集
    private List<String>userActions;
    private final SimpleDateFormattimeNumSdf
                        = new SimpleDateFormat ("yyyyMMddHHmmss");

    public AdminNodeService(){
        this.init();
    }

    private void init(){
        if(jobNodeHosts!=null){
            jobNodeClientMap
                = new ConcurrentHashMap<Integer, LrtdcClient>();
            for(int i=0; i<jobNodeNum; i++){
                String[] tmpHostIp = jobNodeHosts[i].split(":");
                jobNodeClientMap.put(i,
                    new LrtdcClient(tmpHostIp[0],
                    Integer.parseInt (tmpHostIp[1])));
            }
        }
    }
    //AN 任务调度器每次分配的加工成 json 格式的数据列表
    public void setUserActions(List<String>uLogs){
        this.userActions = uLogs;
    }

    //获取当前时间，转换为固定格式“年月日时分秒”长整数
    public long getCurrentTime(){
        return Long.parseLong(timeNumSdf.format(new Date()));
    }
```

```
//获取当前AN下属可用的JN列表
public List<Integer>getAbleJobNodeIds(){
    List<Integer>ableJobNodeIds = new ArrayList<>();
    for(Entry<Integer,LrtdcClient> item
                    : jobNodeClientMap.entrySet()){
        //调用底层通信框架
        if(item.getValue().getHealthStatus()==1){
            ableJobNodeIds.add(item.getKey());
        }
    }
    return ableJobNodeIds;
}

private Random rand = new Random();

public void run(){
    long begin = 0, end = 0;
    int uidNum = userActions.size();

    if(uidNum>0){
        System.out.println(ConfigProperty.getCurDateTime()
            +" 获取最近待执行 " + Constants.rtcPeriodSeconds
            +" 秒用户行为数据共有 "+userActions.size());
        Map<Integer,Integer>rtMap
                        = new HashMap<Integer, Integer>();
        int adminNodeId = -1;
        int rtResult = -1;
        List<Integer>enableJobIds = this.getAbleJobNodeIds();
        int enableServerNum = enableJobIds.size();
        //当前数据集数据个数如果小于系统设定批次任务数据集最小值
        if(uidNum<Constants.minJobBatchNum){
            //随机选一个可用JN进行相关计算
            adminNodeId = enableJobIds.get(
                            rand.nextInt (enableServerNum));
            if(jobNodeClientMap.get(adminNodeId)
                            .getHealthStatus()==1){
            rtResult = jobNodeClientMap.get(adminNodeId)
                        .getRtcStatsResult(userActions);
            //如果计算任务失败，二次重新计算
            if(rtResult<1){
                rtResult = jobNodeClientMap.get(adminNodeId)
                            .getRtcStatsResult(userActions);
                }
            }
            rtMap.put(adminNodeId, rtResult);
        }else{
            //当前数据集数据个数如果大于系统设定批次任务数据集最小值
            List<Future<Map<Integer,Integer>>>futList
            = new LinkedList<Future<Map <Integer,Integer>>>();
            //启动计算任务线程个数为可用JN节点个数
            ExecutorService exPool
             = Executors.newFixedThreadPool(enableServerNum);
```

```
        int eachJobUserNum = (int)Math.ceil(1.0
                * userActions.size()/enableServerNum);
        int startCur=0,endCur=0;
        //批次任务提交
        for(int j=0; j<enableServerNum; j++){
            startCur = j * eachJobUserNum;
            if(j<enableServerNum-1){
                endCur = (j+1) * eachJobUserNum;
                if(endCur>uidNum){
                    endCur = uidNum;
                }
                if(startCur<endCur){
                    futList.add(exPool.submit(
                        new StatsJob(userActions
                        .subList (startCur, endCur), j)));
                }
            }else{
                if(startCur<uidNum){
                    futList.add(exPool.submit(
                        new StatsJob(userActions
                        .subList (startCur, uidNum), j)));
                }
            }
        }
        try {
            //获取任务就按结果
            for(Future<Map<Integer,Integer>> fut
                                                : futList){
                rtMap.putAll(fut.get());
            }
            futList.clear();
            futList = null;
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e){
            e.printStackTrace();
        }
        if(!exPool.isShutdown()){
            exPool.shutdown();
        }
        exPool = null;
        //调用二次计算结果
        this.reRun(rtMap, eachJobUserNum,
                    uidNum, enableServerNum);
    }
    end = System.currentTimeMillis();
    System.out.println(ConfigProperty.getCurDateTime()
            +" 最近待执行的 "+Constants.rtcPeriodSeconds
        +" 秒用户短期兴趣标签共耗时： "
        +(double)(end-begin)/1000+" 秒执行结果："+rtMap);

}
```

```
    }

    /**
     * 检查任务计算结果是否正常，如果某个JN计算失败，则重新计算该任务模块
     * @paramrtMap
     * @parameachJobUserNum
     * @paramuidNum
     */
    public void reRun(Map<Integer,Integer>rtMap,
            int eachJobUserNum, int uidNum, int enableServerNum){
        int startCur=0,endCur=0;
        for(Entry<Integer,Integer> item : rtMap.entrySet()){
            if(item.getValue()<1){
                startCur = item.getKey() * eachJobUserNum;
                if(item.getKey()<enableServerNum-1){
                    endCur = (item.getKey() + 1) * eachJobUserNum;
                    if(endCur>uidNum){
                        endCur = uidNum;
                    }
                    if(startCur<endCur){
                        rtMap.put(item.getKey(),
                          jobNodeClientMap.get(item.getKey())
                            .getRtcStatsResult(userActions
                                    .subList(startCur, endCur)));
                    }
                }else{
                    if(startCur<uidNum){
                        rtMap.put(item.getKey(),
                            jobNodeClientMap.get(item.getKey())
                           .getRtcStatsResult(userActions
                            .subList(startCur, uidNum)));
                    }
                }
            }
        }
    }
    //内部类实现单个JN计算任务的管理
    protected class StatsJob implements Callable<Map<Integer,
Integer>>{

        public List<String>userNids;
        public int nodeId;

        public StatsJob(List<String>userNids, int jobNodeId){
            this.userNids = userNids;
            this.nodeId = jobNodeId;
        }

        @Override
        public Map<Integer, Integer> call() throws Exception {
            Map<Integer,Integer>rtMap
                    = new HashMap<Integer, Integer>();
```

```
            if(jobNodeClientMap.get(this.nodeId)
                                .getHealthStatus()==1){
                //调用底层框架 API
                int rtResult = jobNodeClientMap.get(this.nodeId)
                                . getRtcStatsResult(userNids);
                if(rtResult<1){
                    rtResult = jobNodeClientMap.get(this.nodeId)
                            . getRtcStatsResult(userNids);
                }
                rtMap.put(this.nodeId, rtResult);
            }
            return rtMap;
        }
    }
}
```

如上述两个完整代码分别详细介绍了 AN 中的任务调度器和任务分发器的实现逻辑。

3.3.3 任务计算 JN

JN 为框架 light_drtc 计算的真正落脚点，根据所接受的来自 AN 分配的数据，根据实际相关业务逻辑，为充分利用单机多线程优势，这里我们采用 Fork/Join 框架，实现具体业务需求。这里需要注意的是，务必充分考虑如何更加合理划分若干互相独立的子任务，且子任务足够小到能快速执行完毕。

此模块，开发者在使用时，灵活度非常高，开发者只需实现框架预先设定的接口：“org.light.rtc.base.JobStatsWindow.java”，其完整代码如下：

```
package org.light.rtc.base;

import java.util.List;

public interface JobStatsWindow {
    /**
     * 根据接受的实时数据流，完成相关实时计算业务需求。
     * 根据接受的实时数据流，完成相关实时计算业务需求。
     * @paramuserLogs
     * @return
     */
    int rtcStats(List<String>userLogs);
    /**
     * 适合做线下离线批量处理计算
     * @param start
     * @param size
```

```
     * @return
     */
    int batchStats(int start, int size);
    /**
     * 返回服务健康状态
     * @return
     */
    int getHealthStatus();
}
```

具体开发实例，可以参考测试包中：

- org.light.ldrtc.jober.JobService.java：计算服务相关接口实现。
- org.light.ldrtc.jober.StatsTask.java：任务具体实现类继承 fork/join。

3.4 总结

本章首先向读者介绍了自研框架 light_drtc 的设计思想、逻辑架构和物理架构，并结合当前主流框架 Storm 做了对比，让读者对框架有一个整体认识；其次对于自研框架的三个核心部分 CN、AN 和 JN 的详细设计及核心技术实现相关代码都做了说明，并向读者做了几个核心 API 说明，以便开发者深入理解，也更好地使用框架。

接下来的章节：第 4、5、6、7、8 章，作者将会从实际个性化推荐系统中用户画像实时更新的角度，将系统所需相关技术 MQ、NoSql、ES / Solr 以及相关搜索和常用微服务技术框架向读者详细逐一介绍。

4 第4章 消息队列 MQ

消息队列（Message Queue，MQ）技术是分布式应用间交换信息的一种技术。消息队列可驻留在内存或磁盘上，队列存储消息直到它们被应用程序读取走。通过消息队列，应用程序可独立地执行，它们不需要知道彼此的位置，或在继续执行前不需要等待接收程序接收此消息。下面就从 MQ 用途、原理、当前知名开源 MQ 的使用加以说明。

4.1 消息队列使用场景

消息队列中间件是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量过载等问题实现高性能，高可用，可伸缩和最终一致性架构。对何架构或应用来说，消息队列都是一个至关重要的组件，其使用场景[1]如下：

1．异步通信

有些业务不想也不需要立即处理消息，消息队列就此情况提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消

1 使用消息队列的 10 个理由 http://www.oschina.net/translate/top-10-uses-for-message-queue

息就放多少，然后在需要的时候再去处理它们。

2．解耦

解耦可降低工程间的强依赖程度，针对异构系统进行适配。在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。通过消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。当应用发生变化时，可以独立扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

3．冗余

有时处理数据的过程会失败，除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的“插入—获取—删除”的范式中，在把一个消息从队列中删除之前，需要处理系统明确指出该消息已经被处理完毕，从而确保你的数据被安全保存直到你使用完毕。

4．扩展性

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可，不需要改变代码、不需要调节参数，便于分布式扩容。

5．过载保护

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量无法提前预知；如果以为了能处理这类瞬间峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为超负荷的突发请求而完全崩溃。

6．可恢复性

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的

耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息也仍然可以在系统恢复后被处理。

7．顺序保证

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。

8．缓冲

在任何重要的系统中，都会有需要不同的处理时间的元素。消息队列通过一个缓冲层来帮助任务最高效率的执行，该缓冲有助于控制和优化数据流经过系统的速度，以调节系统响应时间。

9．数据流处理

分布式系统产生的海量数据流，如业务日志、监控数据、用户行为等，针对这些数据流进行实时或批量采集汇总，然后进行大数据分析是当前互联网的必备技术，通过消息队列完成此类数据收集是最好的选择。

4.2 消息队列原理

4.2.1 MQ 使用流程

通常客户端和内网都要进行数据传输和交换，客户端先把数据保存到一个暂时的外网服务器，然后等待内网来拉走处理。客户端看做 producer，内网看做 consumer，这就是消息队列的用武之地。

MQ 使用流程[2]如图 4.1 所示，客户端先把数据写入外网服务器的消息队列，然后内网服务器再从消息队列取走数据，消息队列须满足：

2 消息队列使用 http://www.cnblogs.com/ifonly/p/4989800.html

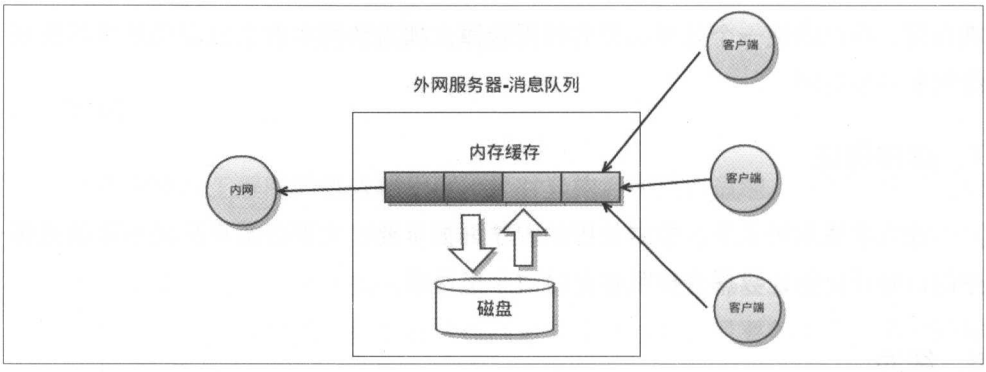


图 4.1 MQ 使用流程

（1）支持高并发的入队和出队操作，允许很多客户端同时插入数据，而且允许内网服务器高频率地取走数据。

（2）支持数据的持久化，为保证速度，消息队列缓存在内存中，如果内存不够用了则写入磁盘。

（3）支持一次性插入/读取多条数据，这样可以提高传输效率。

4.2.2 MQ 基本概念

MQ 常用的一些基本概念[3]如下：

队列管理器：是 MQ 中最上层的一个概念，由它为我们提供消息队列服务。

消息：即应用程序发送给 MQ 托管的数据。其由两部分组成：消息描述符和消息体。消息分为两种类型：永久型和非永久型。

- 永久型：存储在磁盘，在网络和系统发生故障时确保数据不丢失。
- 非永久型：存储在内存，在网络和系统发生故障时数据会丢失。

队列：消息的安全存放地，把消息存放到队列中，直到应用程序或其它 MQ 对象来处理或取走，包括本地/远程/模板队列和动态队列等。

3 MQ 基本概念 http://www.cnblogs.com/guohui-y/p/5201656.html

通道：MQ 队列管理器之间进行通信的渠道。它是建立在物理网络介质上的一个逻辑概念。主要由三种类型：消息通道、MQI 通道和 Cluster 通道。**消息通道**是用于在 MQ 的服务器和服务器之间传输消息的，该通道是单向的，它又有发送者（sender）、接收（receive）、请求者（requestor）、服务者（server）等不同类型，供用户在不同情况下使用。**MQI 通道**是用于 MQ Client 和 MQ Server 之间通信和传输消息的，与消息通道不同，它的传输是双向的。**群集（Cluster）通道**是被用于在位于同一个 MQ 群集内部的队列管理器之间通信的。

侦听器 listener：是接受消息方必不可少的。它侦听 MQ Server 端端口，实时接受异步消息。

4.2.3 MQ 通信模式

MQ 常用的通讯模式[4]主要包括以下四类：

1．点对点通信

点对点方式是最为传统和常见的通信方式，它支持一对一、一对多、多对多、多对一等多种配置方式，支持树状、网状等多种拓扑结构。

2．多点广播

MQ 适用于不同类型的应用。其中重要的，也是正在发展中的“多点广播”应用，即能够将消息发送到多个目标站点（Destination List）。可以使用一条 MQ 指令将单一消息发送到多个目标站点，并确保为每一站点可靠地提供信息。MQ 不仅提供了多点广播的功能，而且还拥有智能消息分发功能，在将一条消息发送到同一系统上的多个用户时，MQ 将消息的一个复制版本和该系统上接收者的名单发送到目标 MQ 系统。目标 MQ 系统在本地复制这些消息，并将它们发送到名单上的队列，从而尽可能减少网络的传输量。

4 MQ 通讯模式 http://www.cnblogs.com/super-d2/p/4541322.html

3. 发布/订阅（Publish/Subscribe）模式

发布/订阅功能使消息的分发可以突破目的队列地理指向的限制，使消息按照特定的主题甚至内容进行分发，用户或应用程序可以根据主题或内容接收到所需要的消息。发布/订阅功能使得发送者和接收者之间的耦合关系变得更为松散，发送者不必关心接收者的目的地址，而接收者也不必关心消息的发送地址，而只是根据消息的主题进行消息的收发。

4. 群集（Cluster）

为了简化点对点通信模式中的系统配置，MQ 提供 Cluster（群集）的解决方案。群集类似于一个域（Domain），群集内部的队列管理器之间通信时，不需要两两之间建立消息通道，而是采用群集（Cluster）通道与其他成员通信，从而大大简化了系统配置。此外，群集中的队列管理器之间能够自动进行负载均衡，当某一队列管理器出现故障时，其他队列管理器可以接管它的工作，从而大大提高系统的高可靠性。

4.2.4 目前知名 MQ 比较

目前互联网行业常用的 MQ 主要包括：Rabbit MQ、Kafka、ZeroMQ、ActiveMQ 和 Redis 等。这里分别介绍[5]如下：

1. RabbitMQ

RabbitMQ 是使用 Erlang 语言编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP，SMTP，STOMP，也正是如此，使得它变得非常有重量级，更适合于企业级的开发，是 AMQP 协议领先的一个实现，它实现了代理（Broker）架构，意味着消息在发送到客户端之前可以在中央节点上排队，对路由（Routing）、负载均衡（Load balance）或者数据持久化都有很好的支持。此特性使得 RabbitMQ 易于使用和部署，适于如路由、负载均衡或消息持久化等很多场景，用消息队列

5 Kafka 深度解析 http://dataunion.org/9307.html

只需几行代码即可搞定。但是，这使得它的可扩展性差，速度较慢，因为中央节点增加了延迟，消息封装后也比较大。如需配置 RabbitMQ 则需要在目标机器上安装 Erlang 环境。

RabbitMQ 的核心概念，如图 4.2 所示，有虚拟主机、交换机、队列、绑定。这里特别强调一点 RabbitMQ 消息模型的核心理念[6]是：是生产者绝对不应该直接将任务投递到目标队列中，换句话说，生产者根本不需要知道任务最终应该会投递到哪里。取而代之的，生产者只需要把任务发送到一个 Exchange 中即可。

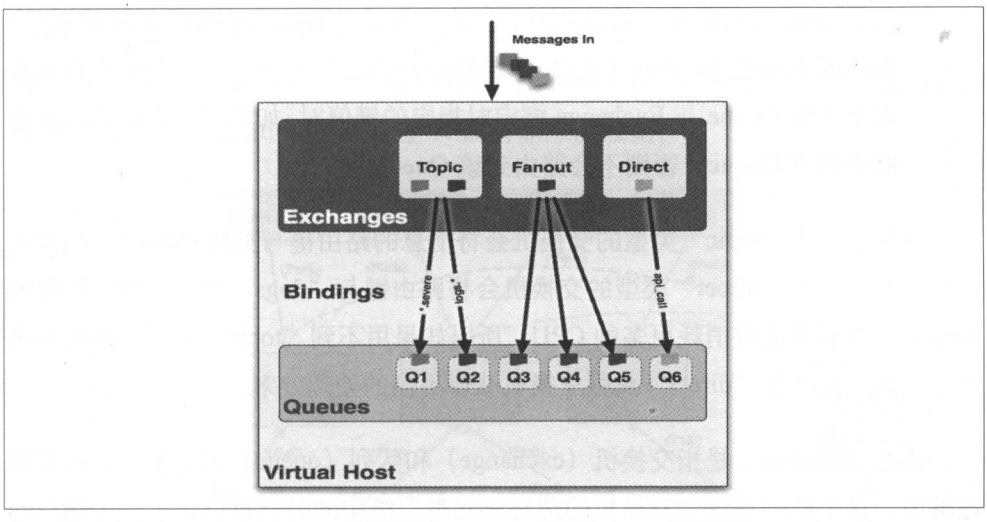


图 4.2 RabbitMQ 核心结构

事实上，发布者（producer）甚至不知道消息是否已经被投递到队列。发布者只需要把消息发送给一个交换机（exchange）。交换机非常简单，它一边从发布者方接收消息，一边把消息推送到队列。交换机必须知道如何处理它接收到的消息，是应该推送到指定的队列还是多个队列，或者是直接忽略消息。这些规则是通过交换机类型（exchange type）来定义的。交换机用来接收消息，转发消息到绑定的队列，共有 4 种类型：Direct，Topic，Fanout 和 Headers，每种规则匹配时的 CPU 开销是不同的，所以根据不同需求选择合适交换机。

6 RabbitMQ 中的交换机 http://www.tuicool.com/articles/Inymqya

- Direct 交换机：转发消息到 routingKey 指定队列（完全匹配，单播）。
- Topic 交换机：按规则转发消息（最灵活，组播），将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。
- Fanout 交换机：转发消息到所有绑定队列（最快，广播），不处理路由键，简单将队列绑定到交换机上，每个发送到交换机的消息都会被转发到与该交换机绑定的所有队列上。
- Headers 交换机：不依赖于 routing key 与 binding key 的匹配规则来路由消息，而是根据发送的消息内容中的 headers 属性进行匹配。在绑定 Queue 与 Exchange 时指定一组键值对；当消息发送到 Exchange 时，RabbitMQ 会取到该消息的 headers（也是一个键值对的形式），对比其中的键值对是否完全匹配 Queue 与 Exchange 绑定时指定的键值对；如果完全匹配消息则会路由到该 Queue，否则不会路由到该 Queue。

例如，一个“topic”类型的交换机会将消息的路由键与类似“dog.*”的模式进行匹配。一个“direct”类型的交换机会将路由键与“dogs”进行比较。匹配末端通配符比直接比较消耗更多的 CPU，所以如果用不到“topic”类型交换机带来的灵活性，就通过“direct”类型交换机获得更高的处理效率。

绑定（binding）是指交换机（exchange）和队列（queue）的关系。可以简单理解为：这个队列对这个交换机的消息感兴趣。绑定的时候可以带上一个额外的 routing_key 参数。为了避免与 basic_publish 的参数混淆，我们把它叫做绑定键（binding key）。绑定键的意义取决于交换机的类型。

2. Jafka/Kafka

Kafka（能将消息分散到不同的节点上）是 LinkedIn 于 2010 年 12 月开发并开源的一个分布式 MQ 系统，现在是 Apache 的一个孵化项目，是一个高性能跨语言分布式 Publish/Subscribe 消息队列系统，而 Jafka 是在 Kafka 基础之上孵化而来的，即 Kafka 的一个升级版。具有以下特性：

- 快速持久化，可以在 O(1)的系统开销下进行消息持久化；
- 高吞吐，在一台普通的服务器上即可达到 10W/s 的吞吐速率；

- 完全分布式系统，Broker、Producer、Consumer 都原生自动支持分布式、实现负载均衡；
- 支持 Hadoop 数据并行加载，对于像 Hadoop 一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka 通过 Hadoop 的并行加载机制来统一在线和离线的消息处理。

此外，对于传统的 message queue 而言，一般会删除已经被消费的消息，而 Kafka 集群会保留所有的消息，无论其是否被消费。当然，因为磁盘限制，不可能永久保留所有数据(实际上也没必要)，因此 Kafka 提供两种策略去删除旧数据：一是基于时间，二是基于 partition 文件大小。例如可以通过配置 $KAFKA_HOME/config/server.properties，让 Kafka 删除一周前的数据，也可通过配置让 Kafka 在 partition 文件超过 1GB 时删除旧数据。

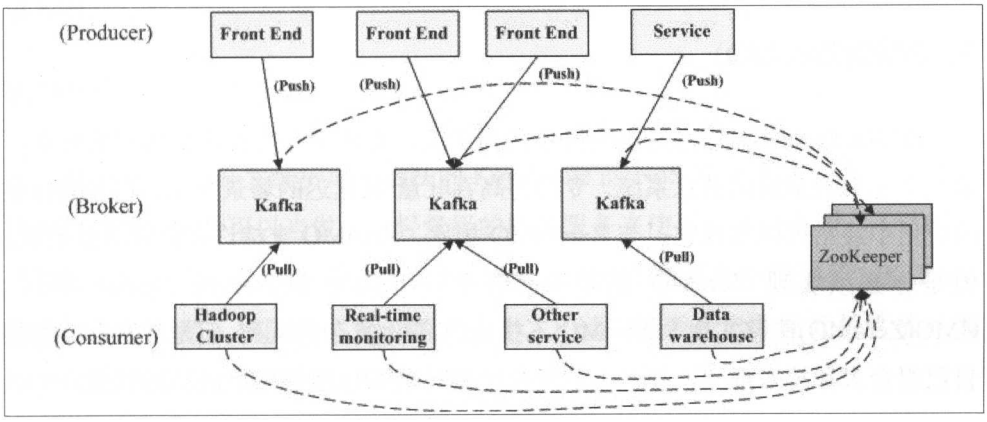


图 4.3 Kafka 部署架构

如图 4.3 所示，一个典型的 Kafka 集群主要由以下几部分组成：

- 若干 **producer**，即负责发布消息到 broker，可以是 Web 前端产生的 page view，或者是服务器日志，系统 CPU、memory 等；
- 若干 **broker**，即 Kafka 集群中包含一个或多个服务器，支持水平扩展，一般 broker 数量越多，集群吞吐率越高；
- 若干 ***Consumer***，即负责消费消息，每个 consumer 属于一个特定的 consumer group（可为每个 consumer 指定 group name，若不指定 group name 则属于

默认的 group)。使用 consumer high level API 时，同一 topic 的一条消息只能被同一个 consumer group 内的一个 consumer 消费，但多个 consumer group 可同时消费这一消息。

- 一个 Zookeeper 集群：Kafka 通过 Zookeeper 管理集群配置，选举 leader，以及在 consumer group 发生变化时进行 rebalance。producer 使用 push 模式将消息发布到 broker，consumer 使用 pull 模式从 broker 订阅并消费消息。

这里重点给大家介绍 Kafka 中的两个概念：Topic 和 Partition。

Topic 在逻辑上可以被认为是一个内在 queue，每条消费都必须指定它的 topic，可以简单理解为必须指明把这条消息放进哪个 queue 里。为了使得 Kafka 的吞吐率可以水平扩展，物理上把 topic 分成一个或多个 partition，每个 partition 在物理上对应一个文件夹，该文件夹下存储这个 partition 的所有消息和索引文件。

3．ØMQ(ZeroMQ)

ØMQ(ZeroMQ)号称最快的消息队列系统，尤其针对大吞吐量的需求场景，是一个非常轻量级的消息系统，专门为高吞吐量/低延迟的场景开发，在金融行业的应用中经常可以发现它。与 RabbitMQ 相比，ZeroMQ 支持许多高级消息场景，但是你必须实现 ZeroMQ 框架中的各个块（比如 Socket 或 Device 等）。ØMQ(ZeroMQ)能够实现 RabbitMQ 不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这 MQ 能够应用成功的挑战。

ZeroMQ 有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色。你只需要简单地引用 ZeroMQ 程序库，可以使用 NuGet 安装，然后你就可以愉快地在应用程序之间发送消息了。但是 ZeroMQ 仅提供非持久性的队列，也就是说，如果宕机，数据将会丢失。其中，Twitter 的 Storm 中使用 ZeroMQ 作为数据流的传输。ZeroMQ 非常灵活，但是你必须学习它长达 80 页的手册。ZeroMQ 没有中间件架构，不需要任何服务进程和运行。事实上，你的应用程序端点扮演了这个服务角色。这让部署起来非常简单，但令人担心的是，你没有地方可以观察它是否有问题出现。就目前了解到的，ZeroMQ 仅提供非持久性的队列。你可以在需要的地方实现自己

的审计和数据恢复功能。

4. Apache ActiveMQ

Apache ActiveMQ 居于 RabbitMQ 和 ZeroMQ 之间，类似于 ZemoMQ，它可以部署于代理模式和 P2P 模式。类似于 RabbitMQ，它易于实现高级场景，而且只需付出低消耗。ActiveMQ 被誉为 Java 世界的中坚力量。它有很长的历史，而且被广泛地使用。它还是跨平台的，给那些非微软平台的产品提供了一个天然的集成接入点。如需配置 ActiveMQ 则需要在目标机器上安装 Java 环境。需要注意一点的是 ActiveMQ 的下一代产品为 **Apollo**，Apollo 以 ActiveMQ 原型为基础，是一个更快、更可靠、更易于维护的消息代理工具。Apache 称 Apollo 为最快、最强健的 STOMP（Streaming Text Orientated Message Protocol，流文本定向消息协议）服务器。

5. Redis

Redis 是一个开发维护均很活跃的 Key-Value 的 NoSQL 数据库，但它本身支持 MQ 功能，完全可以当做一个轻量级的队列服务来使用。实验表明：入队时，当数据比较小时 Redis 的性能要高于 RabbitMQ，而如果数据大小超过了 10K，Redis 则慢得无法忍受；出队时，无论数据大小，Redis 都表现出非常好的性能，而 RabbitMQ 的出队性能则远低于 Redis。

4.3 MQ 消费状态监控

对于 MQ 而言，实时监控 MQ 消费状态有助于我们在生产环境下分析问题，也是 MQ 集群维护非常重要的一个环节。ActiveMQ 和 RabbitMQ 本身所提供的 Web Console 都可以清晰地看到每个消费队列生产和消费 MQ 的情况，而目前在大数据领域炙手可热的 Kafka 本身没有提供专门的 Web 控制台实时监控 MQ 消费状态，幸好 Kafka 开源社区提供了 KafkaOffsetMonitor。

4.3.1 KafkaOffsetMonitor 介绍

KafkaOffsetMonitor 是由 Kafka 开源社区提供的一款 Web 管理界面，这个应用程序用来实时监控 Kafka 服务的 Consumer，以及它们所在的 Partition 中的 Offset，你可以通过浏览当前的消费者组，并且每个 Topic 的所有 Partition 的消费情况都可以观看得一清二楚。它让我们很直观地知道，每个 Partition 的 Message 是否消费掉，有没有阻塞，等等。

这个 Web 管理平台保留的 Partition、Offset 和它的 Consumer 的相关历史数据，我们可以通过浏览 Web 管理的相关模块，清楚地知道最近一段时间的消费情况。

该 Web 管理平台有以下功能：

- 对 Consumer 的消费监控，并列出每个 Consumer 的 Offset 数据。
- 保护消费者组列表信息。
- 每个 Topic 的所有 Partition 列表包含：Topic，Pid，Offset，LogSize，Lag，以及 Owner 等。
- 浏览查阅 Topic 的历史消费信息。

4.3.2 KafkaOffsetMonitor 部署

KafkaOffsetMonitor 的安装部署[7]较为简单，所有的资源都打包到一个 JAR 文件中了，我们可以直接从 https://github.com/quantifind/KafkaOffsetMonitor/releases/latest 上下载最新版的已编译好的最新 Jar 直接运行即可，省去了配置环节。这里我们可以新建一个目录单独用于 Kafka 的监控目录，这里新建的是一个 kafka_monitor 文件目录，然后再准备启动脚本，脚本内容如下所示：

```
java -cp KafkaOffsetMonitor-assembly-0.2.1.jar \
    com.quantifind.kafka.offsetapp.OffsetGetterWeb \
    --offsetStorage kafka  --zk zk-server1,zk-server2  --port 8080
    --refresh 10.seconds  --retain 2.days
```

[7] KafakOffsetMonitor 安装部署 http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=25691489&id=5578732

解释下这条启动命令的含义，首先我们需要指明运行 Web 监控的类，然后需要用到 ZooKeeper，所有要填写 ZK 集群信息，接着是 Web 运行端口，页面数据刷新的时间，以及保留数据的时间值。启动后 Kafka 服务后，我们会在控制台清楚地看到每个 topic 下的 MQ 实时消费情况，如图 4.4.所示。

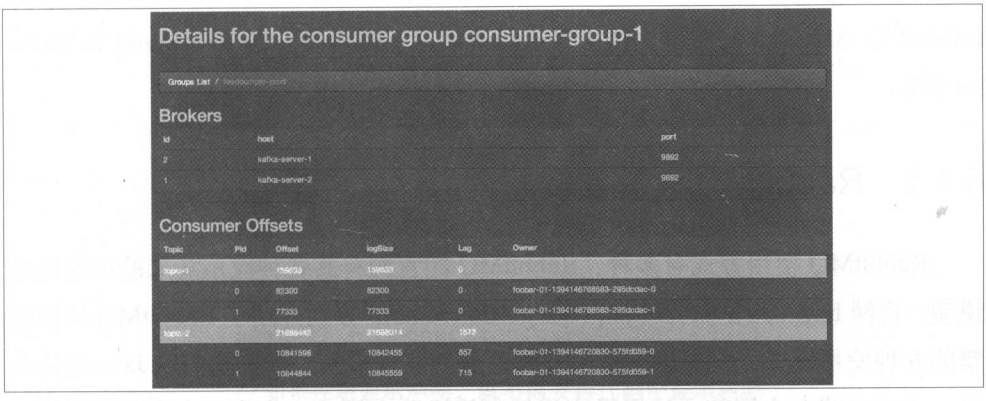


图 4.4 KafkaOffsetMonitor 监控

一些参数的含义如下：

- Topic：创建 Topic 名称。
- Pid：分区编号。
- Offset：表示该 Partition 已经消费了多少 Message。
- LogSize：表示该 Partition 生产了多少 Message。
- Lag：表示有多少条 Message 未被消费。
- Owner：表示消费者。
- Created：表示该 Partition 创建时间。
- Last Seen：表示消费状态刷新最新时间。

Kafka 对 MQ 的 offsets 管理是比较自由和松散的，消费者可以自由选择它目前所支持的三种方式，在 KafkaOffsetMonitor 运行实例时只能选择其中之一存储 Offset。

- 基于 Zookeeper 内置的 high-level 消费；
- 基于 Kafka 内部主题的 kafka 内置的 offset 管理 API；
- Storm-kafka Spout 默认基于 Zookeeper。

4.4 RabbitMQ 和 Kafka 的基本使用

如上节所述，当前主流 MQ 有很多，鉴于篇幅限制，这里不可能一一列举其典型用法，这里只给出当前使用最为广泛的两个 MQ：RabbitMQ 和 Kafka。作者自研框架 light_drtc 所预留的 CN 所需要的 MQ 相关接口，内部实现均提供了对 RabbitMQ 和 Kafka 的支持。这里重点给读者从实例角度介绍 RabbitMQ 和 Kafka 的使用。

4.4.1 RabbitMQ 读写实例

RabbitMQ 使用方式有多种，RabbitMQ 的官网有其各种应用场景的详细使用说明，官网 https://www.rabbitmq.com/getstarted.html 分别介绍了 RabbitMQ 从简到繁的 6 种使用模式，并提供了常用各种语言的实现示例。这里我们以 Java 语言为例，向读者朋友介绍其中使用最广泛的交换机路由模式，作为一个完整的实例，这里向大家介绍消息生产端和消费端的使用说明。

在配置好路由交换机、相关路由 key 及绑定的相应队列（此步骤虽然可以在代码层以硬编码方式绑定，但作者建议使用更加直观的 RabbitMQ 控制台操作）情况下，消息发送端实例：GoodsProduct.java。代码实例以 RabbitMQ3.5.6 为例。项目开始前，pom.xml 中增加 rabbitmq 最新依赖如下：

```
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>3.6.3</version>
</dependency>
```

消息生产者完整源码及说明

```
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

//消息生产者
public class GoodsProduct {
    private final String EXCHANGE_NAME = "userActionEx";//交换机名
    private Connection connection;//mq 连接
```

```
        private Channel channel;//通道
        //时间格式化
        private SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

        public String getCurDateTime(){
            return df.format(new Date());
        }

        public void run(){
            ConnectionFactory factory = new ConnectionFactory();
            factory.setPort(5672);//端口
            factory.setHost("127.0.0.1");//rabbitmq 服务所在 IP
            factory.setUsername("admin");//用户明
            factory.setPassword("light2016");//密码
            factory.setVirtualHost("/bjmq");//虚拟机

            try {
                connection = factory.newConnection();
                channel = connection.createChannel();

                //如果控制显示声明，就不需要再代码中显示声明
    //          channel.exchangeDeclare(EXCHANGE_NAME,      "direct",
true);
                //如果控制显示声明，就不需要再代码中显示声明
    //          channel.queueBind("clickLogQu1",       EXCHANGE_NAME,
"click");
    //          channel.queueBind("clickLogQu2",       EXCHANGE_NAME,
"click");

                String msgStr="test 用户点击新闻 "+this.getCurDateTime();
                for(int i=0; i<10000; i++){
                    try {//间隔 1 秒发送 1 条消息
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    //发送消息
                    channel.basicPublish(EXCHANGE_NAME, "dsssrpc",
                                    null,(msgStr+i).getBytes("UTF-8"));
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                this.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

```
        /**
         * 关闭 channel 和 connection。并非必须，因为隐含是自动调用的。
         * @throws IOException
         */
        public void close() throws IOException{
            this.channel.close();
            this.connection.close();
        }

        public static void main(String[] args) {
            GoodsProduct gp = new GoodsProduct();
            gp.run();
        }
    }
```

以上是 RabbitMQ 的交换机路由模式的消息生产端代码实例说明，有了消息生产源头，自然也需要消息接收端，接下来就是消息消费端：GoodsConsumer.java 代码实例。

```
    import java.io.IOException;
    import java.text.SimpleDateFormat;
    import java.util.Date;
    import java.util.TimerTask;
    import java.util.concurrent.ConcurrentLinkedQueue;

    import com.rabbitmq.client.Channel;
    import com.rabbitmq.client.Connection;
    import com.rabbitmq.client.ConnectionFactory;
    import com.rabbitmq.client.ConsumerCancelledException;
    import com.rabbitmq.client.QueueingConsumer;
    import com.rabbitmq.client.ShutdownSignalException;

    public class GoodsConsumer {
    private final String EXCHANGE_NAME = "userActionEx";//交换机名
        private Connection connection;
        private Channel channel;

        public String getCurDateTime(){
            SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
            return df.format(new Date());
        }

        public void run(){
            ConnectionFactory factory = new ConnectionFactory();
            factory.setPort(5672);//端口
            factory.setHost("127.0.0.1");//rabbitmq 服务所在 IP
            factory.setUsername("admin");//用户明
            factory.setPassword("light2016");//密码
            factory.setVirtualHost("/bjmq");//虚拟机
```

```
            try {
                connection = factory.newConnection();
                channel = connection.createChannel();

                //如果控制显示声明，就不需要再代码中显示声明
    //          channel.exchangeDeclare(EXCHANGE_NAME,
"direct",true);
                //如果控制显示声明，就不需要再代码中显示声明
    //          channel.queueBind("bjActionLog",       EXCHANGE_NAME,
"ccsp");
    //          channel.queueBind("hzClickLog",        EXCHANGE_NAME,
"click");
    //          channel.queueBind("hzSpiderCms",       EXCHANGE_NAME,
"spider");

                //声明消息消费对象工具
                QueueingConsumer consumer = new QueueingConsumer
(channel);
                channel.basicConsume("clickLogQu1", true, consumer);
                channel.basicConsume("clickLogQu2", true, consumer);

                QueueingConsumer.Delivery delivery = null;
                String message = null, routingKey=null;
                while(true){
                    try {
                        delivery = consumer.nextDelivery();
                    } catch (ShutdownSignalException
                            | ConsumerCancelledException
                            | InterruptedException e) {
                        e.printStackTrace();
                    }
                    //获取每条消息的消息体
                    message = new String(delivery.getBody(),"UTF-8");
                    //可获取路由关键词
                    routingKey = delivery.getEnvelope().getRoutingKey();
                    System.out.println(this.getCurDateTime()+ "\t"
                            +message.trim()+"\t"+routingKey);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                this.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        /**
         * 关闭 channel 和 connection。并非必须，因为隐含是自动调用的。
         * @throws IOException
```

```
     */
    public void close() throws IOException{
        this.channel.close();
        this.connection.close();
    }

    public static void main(String[] args) {
        GoodsConsumer gc = new GoodsConsumer();
        gc.run();
    }
}
```

上述实例分别向读者展示了 RabbitMQ 的交换机路由模式使用，该实例通过消息生产端发送若干条消息，消息消费端接收相关信息来展示其使用。这里只是一个最基本的展示，详细使用还需要参考官方。

4.4.2 Kafka 读写实例

很多传统的 message queue 都会在消息被消费完后将消息删除，一方面避免重复消费，另一方面可以保证 queue 的长度比较小，提高效率。Kafka 并不删除已消费的消息，为了实现传统 message queue 消息只被消费一次的语义，Kafka 保证同一个 consumer group 里只有一个 consumer 会消费一条消息。与传统 message queue 不同的是，Kafka 还允许不同 consumer group 同时消费同一条消息，这一特性可以为消息的多元化处理提供了支持。

实际上，Kafka 的设计理念之一就是同时提供离线处理和实时处理。根据这一特性，可以使用 Storm 这种实时流处理系统对消息进行实时在线处理，同时使用 Hadoop 这种批处理系统进行离线处理，还可以同时将数据实时备份到另一个数据中心，只需要保证这三个操作所使用的 consumer 在不同的 consumer group 即可。下面以完整代码实例分别展示 Kafka 作为消息生产者 KafkaProducer.java 和消息消费者 KafkaConsumer.java，代码实例以 Kafka0.8.22 为例。项目开始前，pom.xml 中增加 Kafka 的相关依赖如下：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>0.8.2.2</version>
</dependency>
```

消息生产者 KafkaProducer.java 完整源码：

```
import java.util.Properties;
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class KafkaProducer {
    private final Producer<String, String> producer;
    public final static String TOPIC = "indata_str_
user2news_idfeas";

    private KafkaProducer(){
        Properties props = new Properties();
        //此处配置的是 kafka 的端口
        props.put("metadata.broker.list", "127.0.0.1:9092");
        //配置 value 的序列化类
        props.put("serializer.class",          "kafka.serializer.
StringEncoder");
        //配置 key 的序列化类
        props.put("key.serializer.class",      "kafka.serializer.
StringEncoder");

        //request.required.acks
        //0, which means that the producer never waits for an
acknowledgement from the broker (the same behavior as 0.7). This option
provides the lowest latency but the weakest durability guarantees (some
data will be lost when a server fails).
        //1, which means that the producer gets an acknowledgement
after the leader replica has received the data. This option provides
better durability as the client waits until the server acknowledges the
request as successful (only messages that were written to the now-dead
leader but not yet replicated will be lost).
        //-1, which means that the producer gets an acknowledgement
after all in-sync replicas have received the data. This option provides
the best durability, we guarantee that no messages will be lost as long
as at least one in sync replica remains.
        props.put("request.required.acks","-1");
        //实例化消息生产者
        producer = new Producer<String, String>(new
ProducerConfig(props));
    }

    void produce() {
        int messageNo = 1;
        final int COUNT = 10;
        while (messageNo< COUNT) {
            String key = String.valueOf(messageNo);
            String data = "hello kafka message " + key
                                    +" authord by light";
            //发送每条消息 API：主题、消息 key 和消息内容体
```

```
            producer.send(new KeyedMessage<String, String>(
                                        TOPIC, key ,data));
            messageNo++;
        }
    }

    public static void main( String[] args ){
        KafkaProducerkpd = new KafkaProducer();
        kpd.produce();
    }

}
```

消息消费者 KafkaConsumer.java 完整源码：

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
import kafka.message.MessageAndMetadata;
import kafka.serializer.StringDecoder;
import kafka.utils.VerifiableProperties;

public class KafkaConsumer {
    private final ConsumerConnector consumer;

    private KafkaConsumer() {
        Properties props = new Properties();
        //zookeeper 配置
        props.put("zookeeper.connect",
                  "192.168.10.101:2181,192.168.10.102:2181,"
                  +"192.168.10.103:2181");
        //group 代表一个消费组
        props.put("group.id", "recommend-feature");
        //zk 连接超时
        props.put("zookeeper.session.timeout.ms", "4000");
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");
        props.put("auto.offset.reset", "smallest");//largest
        //序列化类
        props.put("serializer.class",
                  "kafka.serializer. StringEncoder");
        ConsumerConfig config = new ConsumerConfig(props);
        //实例化消息消费者
        consumer = Consumer.createJavaConsumerConnector(config);
```

```
    }

    //消息消费
    void consume() {
        Map<String, Integer>topicCountMap = new HashMap<String,
Integer>();
        //每个 topic 每次获取一个消息
        topicCountMap.put(KafkaProducer.TOPIC, new Integer(1));
        StringDecoder keyDecoder = new StringDecoder(new
    VerifiableProperties());
        StringDecoder valueDecoder = new StringDecoder(new
    VerifiableProperties());
        Map<String,List<KafkaStream<String,String>>>consumerMap
                = consumer.createMessageStreams(topicCountMap,
                  keyDecoder,valueDecoder);
        //获取消息流
        KafkaStream<String, String> stream = consumerMap
                              .get (KafkaProducer.TOPIC).get(0);
        ConsumerIterator<String, String> it = stream.iterator();
        MessageAndMetadata<String, String> meta = null;
        while (it.hasNext()){
            meta = it.next();
            //打印每条消息 key 和消息内容体
            System.out.println(meta.key()+"\t"+meta.message());
        }
    }

    public static void main(String[] args) {
        new KafkaConsumer().consume();
    }

}
```

上述 Kafka 读写完整实例，是 Kafka 的基本使用方式，当然还有很多高级设置，比如 topic 创建、分区设置、旧消息删除等都需要在 Kafka 集群搭建时事先配置好。这里只是抛砖引玉，希望读者在使用时深入学习一下。

4.5 总结

本章主要向读者介绍了 MQ 的使用场景、相关原理、基本概念、通信模型，以及当前知名 MQ 比较，并结合实际，给出了在自研框架 light_drtc 所使用 RabbitMQ，以及 Kafka 作为消息生产者和消费者的完整使用实例，以方便读者全面了解 MQ。

5 第5章 内存数据库 Redis3.0 及 SSDB

内存数据库，顾名思义就是将数据放在内存中直接操作的数据库，它从根本上抛弃了磁盘数据管理的许多传统方式，基于全部数据都在内存中重新设计了体系结构，并且在数据缓存、快速算法、并行操作方面也进行了相应的改进，从而使数据处理速度一般比传统数据库的数据处理速度快很多，一般都会快 10 倍以上。内存数据库从范型上可以分为关系型内存数据库和键值型内存数据库。在实际应用中内存数据库主要配合关系数据库使用，关注性能，并不注重数据完整性和数据一致性。基于键值型的内存数据库比关系型更加易于使用，性能和可扩展性更好，因此在应用上比关系型的内存数据库使用更多。本章也主要向大家介绍键值型内存数据库代表 Reids 及 SSDB。

5.1 Redis 相关介绍

Redis 是一个开源（BSD 许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如字符串、散列、列表、集合、有序集合与范围查询、bitmaps、HyperLogLogs 和地理空间索引半径

查询。Redis 内置了复制、LUA 脚本、LRU 驱动事件、事务和不同级别的磁盘持久化，并通过 Redis 哨兵和自动分区提供高可用性。Redis 的出现，不仅可以用于缓存，也可以用于一些场景的存储，在很多情况下是关系数据库很好的补充。它同时提供了 Python，Ruby，Erlang，PHP 客户端，使用非常方便。

Redis3.0.0 发布于 2015 年 4 月 1 日，之前线上运维大都采用 Version2.8.0，目前最新版 Redis3.0.7。其最重要的新特性是集群（Redis Cluster），提供 Redis 功能子集（比如不支持多数据库）的分布式、容错的实现（最多支持 1000 结点），其自动扩展、容错和高可用性都大大提高。

SSDB 是一个高性能的支持丰富数据结构的 NoSQL，用于替代 Redis。其特点是基于文件存储系统所以它支撑量大的数据而不因为内存的限制受取约束。从官网的测试报告来看其性能也非常出色，和 Redis 相当，因此可以使用它来代替 Redis 来进行 k-v 数据业务的处理.

5.1.1 Redis3.0 集群架构

Redis3.0 之前，为在内存中存储大量数据，需要实现数据分片，均采用客户端根据主键 ID 哈希一致算法，完成数据分区存储，在分区高可用和扩展性方面很有局限性。Redis3.0 的出现改变了这一局面，如图 5.1 所示，Reids-Cluster[1]就可以自动完成数据分区。

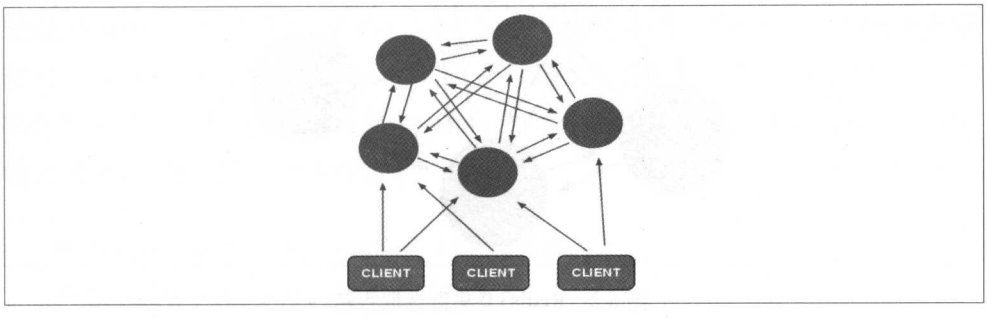


图 5.1 Redis3.0 集群架构

1 redis-cluster 研究和使用 http://hot66hot.iteye.com/blog/2050676

（1）Cluster 采用无中心结构，所有 Redis 节点彼此互联，每个节点都保存数据和整个集群的状态，内部使用二进制协议优化传输速度和带宽。

（2）节点的 fail 是通过集群中超过半数的节点检测失效时才生效的。

（3）客户端与 Redis 节点直连，不需要中间 proxy 层。客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可。

（4）Cluster 预分好 16384 个桶（slot），根据 CRC16(key) mod 16384 的值，决定将一个 key 放到哪个桶中，cluster 负责维护 node<->slot<->key。每个 Redis 物理结点负责一部分桶的管理，当发生 Redis 节点的增减时，调整桶的分布即可。

5.1.2 Redis3.0 集群选举与容错

为了保证服务的可用性，Redis Cluster 采取的方案是 Master-Slave。每个 Redis Node 可以有一个或者多个 Slave。当 Master 挂掉时，选举一个 Slave 形成新的 Master。一个 Redis Node 包含一定量的桶，当这些桶对应的 Master 和 Slave 都挂掉时，这部分桶对应的数据不可用。

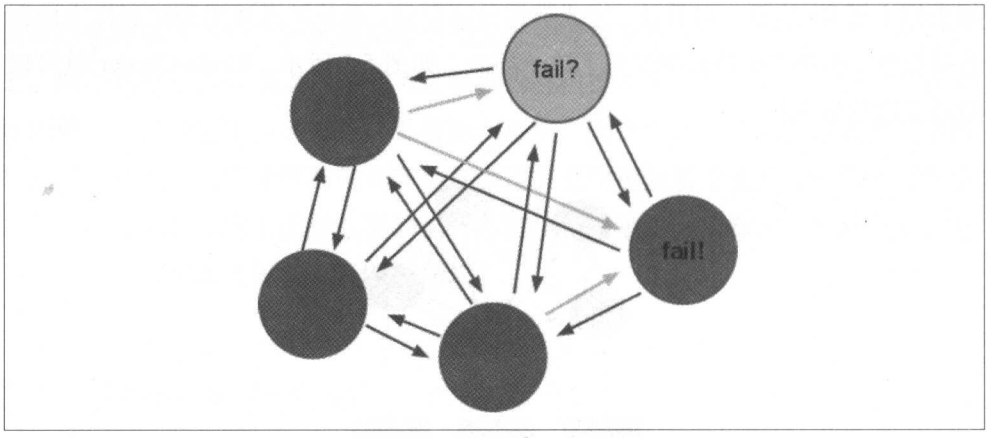


图 5.2 Redis3.0 集群选取容错

Cluster 支持在线增/减节点。基于桶的数据分布方式大大降低了迁移成本，只需将数据桶从一个 Redis Node 迁移到另一个 Redis Node 即可。当桶从一个 Node A

向另一个 Node B 迁移时，Node A 和 Node B 都会有这个桶，Node A 上桶的状态设置为 MIGRATING，Node B 上桶的状态被设置为 IMPORTING。当客户端请求时，所有在 Node A 上的请求都将由 A 来处理，所有不在 A 上的 key 都由 Node B 来处理。同时，Node A 上将不会创建新的 key。具体如图 5.2 所示。

（1）随着选举过程由集群中所有 master 均参与了，如果半数以上 master 节点与 master 节点通信超过（cluster-node-timeout），认为当前 master 节点挂掉。

（2）什么时候整个集群不可用（cluster_state:fail）？

- 如果集群任意 master 挂掉，且当前 master 没有 slave。集群进入 fail 状态，也可以理解成集群的 slot 映射[0-16383]不完成时进入 fail 状态。
- Redis3.0 加入 cluster-require-full-coverage 参数，默认关闭，打开集群兼容部分失败。
- 如果集群超过半数以上 master 挂掉，无论是否有 slave 集群进入 fail 状态。
- 当集群不可用时，所有对集群的操作都不可用，收到((error) CLUSTERDOWN The cluster is down)错误。

5.1.3 SSDB 简介

SSDB 是一个 C/C++语言开发的高性能 NoSQL 数据库[2]，支持 zset(sorted set)、map(hash)、kv、list 等数据结构，用来替代或者与 Redis 配合存储十亿级别列表的数据。SSDB 在 QIHU 360 被大量使用，同时也被国内外业界的众多互联网企业所使用。SSDB 支持在线备份功能，可以通过网络备份数据，不再担心数据丢失。SSDB 还支持主从同步复制（Replication），可用于负载均衡。目前尚不支持服务端的 Shard 数据分区。其特性如下：

- 替代 Redis 数据库，Redis 的 100 倍容量。
- LevelDB 网络支持，使用 C/C++开发。
- Redis API 兼容，支持 Redis 客户端。

2 高性能 No SQL 数据库 SSDB http://www.oschina.net/p/ssdb

- 适合存储集合数据，如 list, hash, zset 等。
- 客户端 API 支持的语言包括：C++、PHP、Python、Cpy、Java、NodeJS、Ruby、Go。
- 内存占用比较少，持久化的队列服务。
- 支持主从同步，多主同步复制，负载均衡。

图 5.3 是 SSDB 官方所提供的对比效果图，从中可以看出，SSDB 和 Redis 相比，在写上略差，在读上略好，当然也有不同意见的实验结论。虽然目前 SSDB 尚不是特别完善，至少这里可以证明 SSDB 是一款性能不错的 NoSQL，值得大家参考学习。

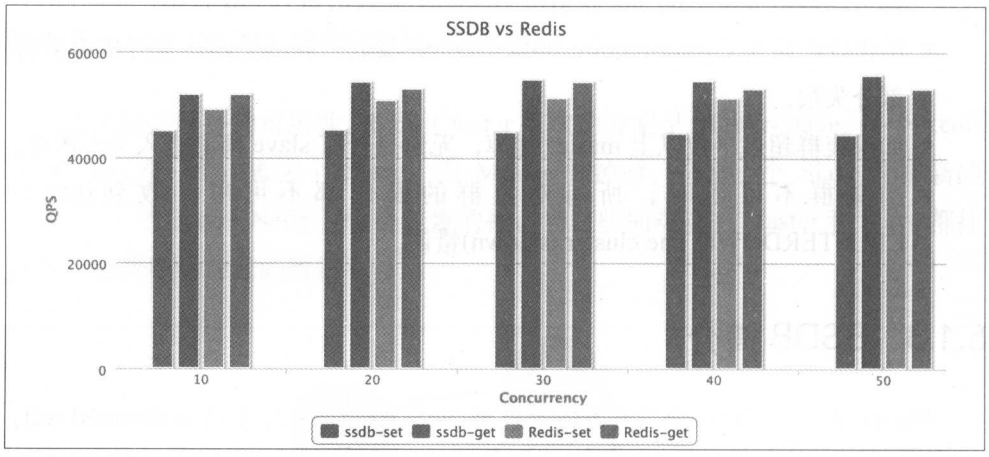


图 5.3 SSDB vs Redis 性能官方对比效果

5.2 Redis3.0 集群搭建

作者曾在 Debian 7 上搭建过 Redis3.0.5 的集群，要让集群保证 HA 正常工作至少需要 3 个主节点，在这里我们要创建含 3 个 shard 的 Redis 集群，6 个 Redis 节点，其中三个是主节点，三个是从节点。其中下载 Redis3.0.5.tar.gz 放到指定的规划好的 Redis 相关数据目录，修改好 Redis 的基本配置文件 redis.conf，并启动 Redis 服务实例。下面重点说下三个核心步骤。

5.2.1 集群所依赖的 Ruby 环境

(1) 以如下方式脚本安装 Ruby 环境：

```
root@classb-recomd-4:~#  apt-get install ruby
Reading package lists... Done
Building dependency tree
Reading state information... Done
ruby is already the newest version.
The following package was automatically installed and is no longer
required:
libnuma1
Use 'apt-get autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 37 not upgraded.
```

(2) 安装 rubygems 组件：

```
root@classb-recomd-4:~# gem install redis
Fetching: redis-3.2.1.gem (100%)
Successfully installed redis-3.2.1
1 gem installed
Installing ri documentation for redis-3.2.1...
Installing RDoc documentation for redis-3.2.1...
```

5.2.2 Redis 集群创建

上述基本环境配置好之后，开始正式搭建 Redis 集群（事先启动各个 Redis 节点服务）。

```
root@classb-recomd-3:/data/redis/redis-3.0.5#  src/redis-trib.rb
create    --replicas    1    10.165.145.33:6379    10.165.145.33:6479
10.165.145.32:6379       10.165.145.32:6479       10.165.145.31:6379
10.165.145.31:6479
>>> Creating cluster
Connecting to node 10.165.145.33:6379: OK
Connecting to node 10.165.145.33:6479: OK
Connecting to node 10.165.145.32:6379: OK
Connecting to node 10.165.145.32:6479: OK
Connecting to node 10.165.145.31:6379: OK
Connecting to node 10.165.145.31:6479: OK
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
10.165.145.33:6379
10.165.145.32:6379
10.165.145.31:6379
Adding replica 10.165.145.32:6479 to 10.165.145.33:6379
Adding replica 10.165.145.33:6479 to 10.165.145.32:6379
```

```
Adding replica 10.165.145.31:6479 to 10.165.145.31:6379
M: fbb8d9b9c9cf7511632874a1f40d6a6d332f8c47 10.165.145.33:6379
   slots:0-5460 (5461 slots) master
S: 7f955e7727cdab230bf1f6d2292640b50b5a2f61 10.165.145.33:6479
   replicates 558a5231158bb4e953a5992dc6af50d191b7399c
M: 558a5231158bb4e953a5992dc6af50d191b7399c 10.165.145.32:6379
   slots:5461-10922 (5462 slots) master
S: 213114eef95d5ac9020f1f714bfe8b8ab974bb63 10.165.145.32:6479
   replicates fbb8d9b9c9cf7511632874a1f40d6a6d332f8c47
M: 1a062ad16a7240e135fee5a05a7890c38de3dc5a 10.165.145.31:6379
   slots:10923-16383 (5461 slots) master
S: 4971516905b757bb0097d189eee18490056a2dab 10.165.145.31:6479
   replicates 1a062ad16a7240e135fee5a05a7890c38de3dc5a
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join..
>>> Performing Cluster Check (using node 10.165.145.33:6379)
M: fbb8d9b9c9cf7511632874a1f40d6a6d332f8c47 10.165.145.33:6379
   slots:0-5460 (5461 slots) master
M: 7f955e7727cdab230bf1f6d2292640b50b5a2f61 10.165.145.33:6479
   slots: (0 slots) master
   replicates 558a5231158bb4e953a5992dc6af50d191b7399c
M: 558a5231158bb4e953a5992dc6af50d191b7399c 10.165.145.32:6379
   slots:5461-10922 (5462 slots) master
M: 213114eef95d5ac9020f1f714bfe8b8ab974bb63 10.165.145.32:6479
   slots: (0 slots) master
   replicates fbb8d9b9c9cf7511632874a1f40d6a6d332f8c47
M: 1a062ad16a7240e135fee5a05a7890c38de3dc5a 10.165.145.31:6379
   slots:10923-16383 (5461 slots) master
M: 4971516905b757bb0097d189eee18490056a2dab 10.165.145.31:6479
   slots: (0 slots) master
   replicates 1a062ad16a7240e135fee5a05a7890c38de3dc5a
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

5.2.3 Redis 集群验证

验证 Redis 集群是否搭建好的方式如下，下属命令验证集群节点。

```
appops@classb-recomd-4:~$ redis-cli cluster nodes
7f955e7727cdab230bf1f6d2292640b50b5a2f61 10.165.145.33:6479 slave
558a5231158bb4e953a5992dc6af50d191b7399c 0 1446518880003 3 connected
fbb8d9b9c9cf7511632874a1f40d6a6d332f8c47      10.165.145.33:6379
master - 0 1446518881003 1 connected 0-5460
1a062ad16a7240e135fee5a05a7890c38de3dc5a      10.165.145.31:6379
master - 0 1446518877998 5 connected 10923-16383
```

```
    213114eef95d5ac9020f1f714bfe8b8ab974bb63 10.165.145.32:6479 slave
fbb8d9b9c9cf7511632874a1f40d6a6d332f8c47 0 1446518879000 4 connected
    4971516905b757bb0097d189eee18490056a2dab 10.165.145.31:6479 slave
1a062ad16a7240e135fee5a05a7890c38de3dc5a 0 1446518875992 6 connected
    558a5231158bb4e953a5992dc6af50d191b7399c       10.165.145.32:6379
myself,master - 0 0 3 connected 5461-10922
```

验证 Redis 集群 Slot 命令如下：

```
appops@classb-recomd-5:~$ redis-cli -p 6479
127.0.0.1:6479> cluster slots
1) 1) (integer) 10923
   2) (integer) 16383
   3) 1) "10.165.145.31"
      2) (integer) 6379
   4) 1) "10.165.145.31"
      2) (integer) 6479
2) 1) (integer) 0
   2) (integer) 5460
   3) 1) "10.165.145.33"
      2) (integer) 6379
   4) 1) "10.165.145.32"
      2) (integer) 6479
3) 1) (integer) 5461
   2) (integer) 10922
   3) 1) "10.165.145.32"
      2) (integer) 6379
   4) 1) "10.165.145.33"
      2) (integer) 6479
```

5.2.4 SSDB 简单部署

SSDB 是值得大家学习的 NoSQL，因此，笔者这里也给大家介绍下 SSDB1.9.2 安装步骤。因为 SSDB 目前上不支持 Shard，笔者只是在自己电脑上安装。

首先从官网下载最新版 https://github.com/ideawu/ssdb/archive/master.zip，解压后进入其根目录，执行 make 编译，如果没有问题，直接安装。编译信息太多，这里只给出编译无误后的安装记录。

（1）SSDB 安装流程。

```
localhost:ssdb-master a$ sudo make install
Password:
mkdir -p /usr/local/ssdb
mkdir -p /usr/local/ssdb/_cpy_
mkdir -p /usr/local/ssdb/deps
```

```
mkdir -p /usr/local/ssdb/var
mkdir -p /usr/local/ssdb/var_slave
cp -f ssdb-server ssdb.conf ssdb_slave.conf /usr/local/ssdb
cp -rf api /usr/local/ssdb
cp -rf \
        tools/ssdb-bench \
        tools/ssdb-cli tools/ssdb_cli \
        tools/ssdb-cli.cpy tools/ssdb-dump \
        tools/ssdb-repair \
        /usr/local/ssdb
cp -rf deps/cpy /usr/local/ssdb/deps
chmod 755 /usr/local/ssdb
chmod -R ugo+rw /usr/local/ssdb/*
rm -f /usr/local/ssdb/Makefile
```

SSDB 安装后，默认安装在系统的 / usr/local/ssdb 目录。接下来就是启动 ssdb 服务，并通过客户端体验基本操作。

（2）ssdb 服务启动。

```
deMacBook-Pro-62:ssdb a$ sudo ./ssdb-server -d ssdb.conf
Password:
ssdb-server 1.9.2
Copyright (c) 2012-2015 ssdb.io
```

（3）SSDB 客户端基本操作，和 Redis 兼容。

```
adeMacBook-Pro-62:ssdb a$ ./ssdb-cli -h 127.0.0.1 -p 8888
ssdb (cli) - ssdb command line tool.
Copyright (c) 2012-2015 ssdb.io
'h' or 'help' for help, 'q' to quit.
ssdb-server 1.9.2
ssdb 127.0.0.1:8888> set k light20160430
ok
(0.001 sec)
ssdb 127.0.0.1:8888> get k
light20160430
(0.000 sec)
ssdb 127.0.0.1:8888> del k
ok
(0.001 sec)
ssdb 127.0.0.1:8888> get k
not_found
(0.000 sec)
```

上述基本操作正是演示了设置键值对<k,v>，以及获取键 k、删除键 k 等基本操作。

5.3 Redis 管理及使用

Redis 目前业界使用非常广泛，Redis 中国社区也很活跃，其中有丰富的使用文档[3]可供参考。这里从使用角度向读者介绍其中一些关键点。

5.3.1 Redis 基本使用

1. Redis命令

Redis 完整的命令列表，常用命令如下：

```
增（改）查命令：set key value, del key,  get key。
keys * 取出当前所有的 key
exists name 查看 n 是否有 name 这个 key
del name 删除 key name
expire confirm 100 设置 confirm 这个 key100 秒过期
ttl confirm 获取 confirm 这个 key 的有效时长
select 0 选择到 0 数据库 redis 默认的数据库是 0~15 一共 16 个数据库
```

move confirm 1 将当前数据库中的 key 移动到其他的数据库中，这里就是把 confirm 这个 key 从当前数据库中移动到 1 中。

```
persist confirm 移除 confirm 这个 key 的过期时间
randomkey 随机返回数据库里面的一个 key
rename key2 key3 重命名 key2 为 key3
type key2 返回 key 的数据类型
```

详细信息参考：http://www.redis.cn/commands.html。

2. 管道（Pipelining）

Redis 是一种基于客户端—服务端模型，以及请求/响应协议的 TCP 服务。一次请求/响应服务器能实现处理新的请求，即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。这

3 Redi 中国社区使用文档 http://www.redis.cn/documentation.html

就是管道（pipelining），是一种几十年来广泛使用的技术。详细信息参考：http://www.redis.cn/topics/pipelining.html。

3．Redis发布/订阅（Pub/Sub）

订阅、取消订阅和发布这三个操作实现了发布/订阅消息范式。发送者（发布者）不是计划给特定的接收者（订阅者）发送消息，而是把要发布的消息分到不同的频道，不需要知道什么样的订阅者订阅。订阅者对一个或多个频道感兴趣，只需接收感兴趣的消息，不需要知道什么样的发布者发布的。这种发布者和订阅者的解耦合可以带来更大的扩展性和更加动态的网络拓扑。Redis 同时支持匹配模式和频道订阅，详细信息参考：http://www.redis.cn/topics/pubsub.html。

4．RedisLua脚本

Redis 2.6 Lua 脚本相关文档可参见：http://www.redis.cn/commands/eval.html。

5．过期（Expires）

Redis 允许为每一个 key 设置不同的过期时间，当它们到期时将自动从服务器上删除。详细文档参考：http://www.redis.cn/commands/expire.html。

6．大量插入数据

从 Redis 2.6 开始 redis-cli 支持一种新的被称之为 pipe mode 的新模式，用于执行大量数据插入工作。使用 pipe mode 模式的执行命令如下：

```
cat data.txt | redis-cli -pipe
```

这将产生类似如下的输出：

```
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 1000000
```

pipe mode 的工作原理是：

- redis-cli pipe 试着尽可能快地发送数据到服务器。
- 读取数据的同时解析它。
- 一旦没有更多的数据输入，它就会发送一个特殊的 ECHO 命令，后面跟着 20 个随机的字符。我们相信可以通过匹配回复相同的 20 个字符是同一个命令的行为。
- 一旦这个特殊命令发出，收到的答复就开始匹配这 20 个字符，当匹配时，就可以成功退出了。
- 同时，在分析回复的时候，我们会采用计数器的方法计数，以便在最后能够告诉我们大量插入数据的数据量。

7．分区（Partitioning）

如何将你的数据分布在多个 Redis 里面，之前依赖客户端，现在只要 Redis 服务集群本身即可。

5.3.2 Redis 管理

1．Redis事务

MULTI、EXEC、DISCARD 和 WATCH 是 Redis 事务相关的命令。事务可以一次执行多个命令，并且带有以下两个重要的保证：

- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

EXEC 命令负责触发并执行事务中的所有命令：

- 如果客户端在使用 MULTI 开启了一个事务之后，却因为断线而没有成功执行 EXEC，那么事务中的所有命令都不会被执行。
- 另一方面，如果客户端成功在开启事务之后执行 EXEC ，那么事务中的所有命令都会被执行。

详细信息参考：http://www.redis.cn/topics/transactions.html。

2. 内存回收

Redis 被当做缓存来使用，当你新增数据时，让它自动地回收旧数据是件很方便的事情。这个行为在开发者社区非常有名，因为它是流行的 memcached 系统的默认行为。LRU 是 Redis 唯一支持的回收方法。当 maxmemory 限制达到的时候 Redis 会使用的行为由 Redis 的 maxmemory-policy 配置指令来进行配置。主要策略如下：

- **noeviction**：返回错误，当达到内存限制并且客户端尝试执行会让更多内存被使用的命令（比如，大部分的写入指令，但 DEL 命令例外）。
- **allkeys-lru**：尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。
- **volatile-lru**：尝试回收最少使用的键（LRU），但仅限于在过期集合的键，使得新添加的数据有空间存放。
- **allkeys-random**：回收随机的键使得新添加的数据有空间存放。
- **volatile-random**：回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。
- **volatile-ttl**：回收在过期集合的键，并且优先回收存活时间（TTL）较短的键，使得新添加的数据有空间存放。

选择正确的回收策略是非常重要的，这取决于应用的访问模式，一般的经验规则是：

- 使用 allkeys-lru 策略：当你希望你的请求符合一个幂定律分布，也就是说，你希望部分的子集元素将比其他元素被访问的更多。当不确定选择什么，这是个很好的选择。
- 使用 allkeys-random：如果是循环访问，所有的键被连续扫描，或者希望请求分布正常（所有元素被访问的概率都差不多）。
- 使用 volatile-ttl：如果你想要通过创建缓存对象时设置 TTL 值，来决定哪些对象应该被过期。

详细文档参考：http://www.redis.cn/topics/memory-optimization.html。

3．配置（Configuration）

Redis 可以在没有配置文件的情况下通过内置的配置来启动，但是这种启动方式只适用于开发和测试。合理的配置 Redis 的方式是提供一个 Redis 配置文件，这个文件通常叫做 redis.conf。详细信息参考：http://www.redis.cn/topics/config.html。

4．复制（Replication）

Redis 复制很简单易用，它通过配置允许 slave Redis Servers 或者 Master Servers 的复制品。详细信息参考：http://www.redis.cn/topics/replication.html。

5．持久化（Persistence）

Redis 提供了不同级别的持久化方式：

- RDB 持久化方式能够在指定的时间间隔对数据进行快照存储。
- AOF 持久化方式记录每次对服务器写的操作，当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF 命令以 Redis 协议追加保存每次写的操作到文件末尾。Redis 还能对 AOF 文件进行后台重写，使得 AOF 文件的体积不至于过大。
- 如果只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。
- 你也可以同时开启两种持久化方式，在这种情况下，当 Redis 重启的时候会优先载入 AOF 文件来恢复原始的数据，因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集完整。

一般来说，如果想达到足以媲美 PostgreSQL 的数据安全性，你应该同时使用两种持久化功能。如果你非常关心你的数据，但仍然能承受数分钟以内的数据丢失，那么可以只使用 RDB 持久化。有很多用户都只使用 AOF 持久化，但我们并

不推荐这种方式：因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份，并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度快，除此之外，使用 RDB 还可以避免之前提到的 AOF 程序的 Bug。

详细信息参考：http://www.redis.cn/topics/persistence.html。

6．安全性（Security）

Redis 提供的访问控制，代码安全问题，通过外部的恶意输入触发的攻击和其他类似的主题，详细文档参考：http://www.redis.cn/topics/security.html。

7．连接处理（Connections Handling）

当客户端初始化后，Redis 检查我们是否还在它可以同时处理的客户端的数量限制范围内。这是使用 maxclients 配置指令配置的，如果它因为当前已经接受了最大数量的客户端，无法接受当前的客户端，Redis 将尝试发送一个错误给客户端以便让其意识到这种情况，并且立即关闭连接。即使连接被 Redis 立即关闭，错误信息也会返回给客户端，因为新的 socket 输出缓冲区一般情况下都足够放下错误信息，因而客户端内核将处理连接错误。详细文档参考：http://www.redis.cn/topics/clients.html。

8．高可用性（High Availability）

Redis Sentinel 是 Redis 官方的高可用性解决方案。目前已经可用。

5.4 Redis 客户端应用

5.4.1 Redis3.0 客户端

Redis 客户端有很多，这里主要介绍官方所提供的、广为使用的 Jedis，应对 Redis3.0，一般采用 Jedis 的 Version2.7.2 之后版本。如下所示，只需在项目中引入

依赖即可。

```
    <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>2.8.1</version>
        <type>jar</type>
        <scope>compile</scope>
    </dependency>
```

下面给出 Jedis 操作 Redis 的基本使用方法说明，下例为完整实例。

```
    import java.util.HashMap;
    import java.util.Map;
    import java.util.Map.Entry;
    import java.net.InetAddress;
    import java.net.UnknownHostException;
    import java.util.ArrayList;
    import java.util.Set;

    import org.light.rtc.util.Constants;

    import net.sf.json.JSONObject;
    import redis.clients.jedis.Jedis;
    import redis.clients.jedis.JedisShardInfo;
    import redis.clients.jedis.Pipeline;
    import redis.clients.jedis.Response;
    import redis.clients.jedis.ShardedJedis;

    public class RedisDao {
        //单例模式 DAO
        private static RedisDao instance;
        //集群使用模式
    //  private ShardedJedisjedis;
        //普通模式
        private Jedisjedis;

        private RedisDao(){
            this.ncrInit();
        }

        public void ncrInit(){
            //redis 集群模式客户端初始化方法
    //      String[] redisList = Constants.redisHost.split(";");
    //      ArrayList<JedisShardInfo>       userMasterWriteJedisShard
InfoList = new ArrayList<JedisShardInfo>();
    //       for (int i = 0; i <redisList.length; i++) {
    //          String[] hostPorts = redisList[i].split(":");
    //          JedisShardInfo userMasterWriteJedis = null;
    //          try {
    //              userMasterWriteJedis = new JedisShardInfo(
    //      InetAddress.getByName(hostPorts[0]).getHostAddress(),
```

```
    //                      Integer.parseInt(hostPorts[1]), 10000);
                        //访问权限：密码认证
    //                  userMasterWriteJedis.setPassword
(Constants.redisPswd);
    //              } catch (UnknownHostException e) {
    //                  e.printStackTrace();
    //              }
    //              userMasterWriteJedisShardInfoList.add
(userMasterWriteJedis);
    //          }
    //          this.jedis    =    new    ShardedJedis(userMasterWriteJedis
ShardInfoList, ShardedJedis.DEFAULT_KEY_TAG_PATTERN);
                // 单机版 Redis 使用模式
                String[] hostIp = Constants.redisHost.split(":");
                this.jedis = new Jedis(hostIp[0],
                                Integer.parseInt (hostIp[1]),10000);
                //访问权限：密码认证
                this.jedis.auth(Constants.redisPswd);
            }

            //单例模式获取 Redis 访问 DAO 对象
            public synchronized static RedisDaogetInstance(){
                if(instance==null){
                    instance = new RedisDao();
                }
                return instance;
            }
            //根据热点新闻默认键获取热点新闻 ID
            public String getTopNewsIds(String topNewsIds){
                return this.jedis.get(topNewsIds);
            }
            //根据新闻 ID 获取每个新闻对应的 CTR 信息
            public Map<String,String>getNewsCtr(String docId){
                return this.jedis.hgetAll(docId);
            }

            //预设置的超时时间期限：6 小时，即 60 * 60 * 6 秒
            private int expSeconds = 21600;
            //批量添加用户短期兴趣标签，过期时间为 6 小时
            public void addUserShortTags(Map<String,String>shortTags){
                //管道模式，批量操作提高性能
                Pipeline p = jedis.pipelined();
                for(Entry<String,String> item : shortTags.entrySet()){
                    p.setex(item.getKey(),this.expSeconds,
                                                item.getValue());

                }
                p.sync();
            }

            public void test(){
                this.jedis.set("light201652", "light test redis @五道口");
```

```
        System.out.println("\t"+this.jedis.get("light201652"));
        //验证 Redis 的 pipeline 管道使用
        Map<String,Response<String>>respMap
                    = new HashMap<String, Response<String>>();
        for (int i = 0; i < 10; i++) {
            p.set("light2016_" + i, i + "_light");
            respMap.put("light2016_" + i,p.get("light2016_" + i));
            p.del("light2016_" + i);
        }
        p.sync();// 这段
        for(Entry<String,Response<String>> item
                                        : respMap. entrySet()){
            System.out.println(item.getKey()+"\t"
                                    +item. getValue().get());
        }
    }

    public static void main(String[] args) {
        RedisDaordao = new RedisDao();
        rdao.test();
    }
}
```

上述代码给出了 Redis 基本 API 操作的完整范例，当然更多高级使用方法，还需要读者自行学习研究，这里只是给出了一个基本入门实例，引领大家入门而已。

5.4.2 SSDB 客户端

因为 SSDB 基本操作兼容 Redis，所以，客户端操作 SSDB 可以使用 Jedis，但也可以使用 SSDB 自己特有的客户端 ssdb4j，其使用也比较简单，下面是目前最新版依赖。

```
<dependency>
    <groupId>org.nutz</groupId>
    <artifactId>ssdb4j</artifactId>
    <version>9.2</version>
</dependency>
```

完整使用范例如下：

```
import org.nutz.ssdb4j.spi.SSDB;
import org.nutz.ssdb4j.spi.Response;
import org.nutz.ssdb4j.SSDBs;

public class TestSSDB {
```

```
        public void test(){
            //使用默认方式：默认本机、默认本地端口 8888
    //      SSDB ssdb = SSDBs.simple();
            //推荐使用连接池方式。
            SSDB ssdb = SSDBs.pool("127.0.0.1", 8888, 3000, null);
            // call check() to make sure resp is ok
            System.out.println(ssdb.set("name",  " 王 光 ").check().
asString());
            Response resp = ssdb.get("name");
            if (resp.ok()) {
                System.out.println(resp.asString());
            }
        }

        public static void main(String[] args) {
            TestSSDBtsdb = new TestSSDB();
            tsdb.test();
        }
    }
```

SSDB 的最大好处就是可以不受内存的限制，直接使用文件方式代替内存的扩充，确实有其称赞之处，这里只是一个简单说明，详细资料还需要读者朋友自己研究。

5.5 本地缓存 Guava Cache

随着现代服务器硬件配置越来越高，服务器内存动辄高达 128GB、256G 甚至高达 384G，如果可以充分利用本地缓存，对我们的应用更是一个显著性能优化点（毕竟本地缓存省去了网络开销）。前面重点介绍了网络内存数据库，本节重点介绍本地缓存方案。谈到本地缓存，大家首先想到的可能是 JDK 自身的 ConcurrentHashMap，它简单易用且高效，诚然有自己的使用范围，不过也有一些它自身无法完成的功能点：内存限制、时间过期精确控制等，如果想实现，需要很大开发量。此时，我们就可以考虑 Guava Cache。

5.5.1 认识 Guava Cache

Google Guava 包含了 Google 的 Java 项目许多依赖的库，如：集合（collections）、缓存（caching）、原生类型支持（primitives support）、并发库（concurrency libraries）、

通用注解（common annotations）、字符串处理（string processing）、I/O 等等。本节只介绍其中的缓存部分。

GuavaCache 是一个全内存的本地缓存实现，支持多种缓存过期策略，它提供了线程安全的实现机制。整体上来说 Guava Cache 简单易用，性能好，是本地缓存的不二之选。缓存在很多场景下都是很有用的。如，通过 key 获取一个 value 花费的时间很多，而且获取的次数不止一次的时候，就应该考虑使用缓存。Guava Cache 与 ConcurrentMap 很相似，但也不完全一样。最基本的区别是 ConcurrentMap 会一直保存所有添加的元素，直到显式地移除。而 Guava Cache 为了限制内存占用，通常都设定为自动回收元素。在某些场景下，尽管 LoadingCache 不回收元素，它也会自动加载缓存。

注：如果你不需要 Cache 中的特性，使用 ConcurrentHashMap 有更好的内存效率——但 Cache 的大多数特性都很难基于旧有的 ConcurrentMap 复制，甚至根本不可能做到。

5.5.2 Guava Cache 使用

本节主要向读者介绍 Guava－Cache 的缓存加载、回收、清除等基本操作[4]。

1. 加载

在使用缓存前，首先问自己一个问题：有没有合理的默认方法来加载或计算与键关联的值？如果有的话，你应当使用 CacheLoader。如果没有，或者你想要覆盖默认的加载运算，同时保留“获取缓存-如果没有-则计算”（get-if-absent-compute）的原子语义，你应该在调用 get 时传入一个 Callable 实例。缓存元素也可以通过 Cache.put 方法直接插入，但自动加载是首选的，因为它可以更容易地推断所有缓存内容的一致性。

4 Google Guava 缓存 http://ifeve.com/google-guava-cachesexplained/

LoadingCache 是附带 CacheLoader 构建而成的缓存实现。创建自己的 CacheLoader 通常只需要简单地实现 V load(K key) throws Exception 方法。从 LoadingCache 查询的正规方式是使用 get(K)方法。这个方法要么返回已经缓存的值，要么使用 CacheLoader 向缓存原子地加载新值。由于 CacheLoader 可能抛出异常，LoadingCache.get(K)也声明为抛出 ExecutionException 异常。如果你定义的 CacheLoader 没有声明任何检查型异常，则可以通过 getUnchecked(K)查找缓存；但必须注意，一旦 CacheLoader 声明了检查型异常，就不可以调用 getUnchecked(K)。

getAll(Iterable<?extends K>)方法用来执行批量查询。默认情况下，对每个不在缓存中的键，getAll 方法会单独调用 CacheLoader.load 来加载缓存项。如果批量的加载比多个单独加载更高效，你可以重载 CacheLoader.loadAll 来利用这一点。getAll(Iterable)的性能也会相应提升。CacheLoader.loadAll 的实现可以为没有明确请求的键加载缓存值。例如，为某组中的任意键计算值时，能够获取该组中的所有键值，loadAll 方法就可以实现为在同一时间获取该组的其他键值。

注：getAll(Iterable<? extends K>)方法会调用 loadAll，但会筛选结果，只会返回请求的键值对。

所有类型的 GuavaCache，不管有没有自动加载功能，都支持 get(K, Callable<V>)方法。这个方法返回缓存中相应的值，或者用给定的 Callable 运算并把结果加入到缓存中。在整个加载方法完成前，缓存项相关的可观察状态都不会更改。这个方法简便地实现了“如果有缓存则返回；否则运算、缓存、然后返回”。

2. 缓存回收

一个残酷的现实是，我们几乎一定没有足够的内存缓存所有数据。你必须决定：什么时候某个缓存项就不值得保留了？Guava Cache 提供了三种基本的缓存回收方式：基于容量回收、定时回收和基于引用回收。

（1）基于容量的回收（size-based eviction）

如果要规定缓存项的数目不超过固定值，只需使用 CacheBuilder.maximumSize(long)。缓存将尝试回收最近没有使用或总体上很少使用的缓存项。在缓存项的数目达到限定值之前，缓存就可能进行回收操作——通常来说，这种情况发生在缓存项的数目逼近限定值时。

另外，不同的缓存项有不同的“权重”（weights），例如，如果你的缓存值，占据完全不同的内存空间，你可以使用 CacheBuilder.weigher(Weigher)指定一个权重函数，并且用 CacheBuilder.maximumWeight(long)指定最大总重。在权重限定场景中，除了要注意回收也是在重量逼近限定值时就进行了，还要知道重量是在缓存创建时计算的，更要考虑重量计算的复杂度。

（2）定时回收（Timed Eviction），CacheBuilder 提供两种定时回收的方法。

- expireAfterAccess(long, TimeUnit)：缓存项在给定时间内没有被读/写访问，则回收。请注意这种缓存的回收顺序和基于大小回收一样。
- expireAfterWrite(long, TimeUnit)：缓存项在给定时间内没有被写访问（创建或覆盖），则回收。如果认为缓存数据总是在固定时候后变得陈旧不可用，这种回收方式是可取的。

定时回收周期性地在写操作中执行，偶尔在读操作中执行。

（3）基于引用的回收（Reference-based Eviction）

通过使用弱引用的键、或弱引用的值、或软引用的值，Guava Cache 可以把缓存设置为允许垃圾回收：

- CacheBuilder.weakKeys()：使用弱引用存储键。当键没有其他（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使用弱引用键的缓存用==，而不是 equals 比较键。
- CacheBuilder.weakValues()：使用弱引用存储值。当值没有其他（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使

用弱引用值的缓存用==，而不是 equals 比较值。

- CacheBuilder.softValues()：使用软引用存储值。软引用只有在响应内存需要时，才按照全局最近最少使用的顺序回收。考虑到使用软引用的性能影响，我们通常建议使用更有性能预测性的缓存大小限定（见上文，基于容量回收）。使用软引用值的缓存同样用==而不是 equals 比较值。

3．显式清除

任何时候，你都可以显式地清除缓存项，而不是等到它被回收。

- 个别清除：Cache.invalidate(key)
- 批量清除：Cache.invalidateAll(keys)
- 清除所有缓存项：Cache.invalidateAll()

4．移除监听器

通过 CacheBuilder.removalListener(RemovalListener)，你可以声明一个监听器，以便缓存项被移除时做一些额外操作。缓存项被移除时，RemovalListener 会获取移除通知[RemovalNotification]，其中包含移除原因[RemovalCause]、键和值。

请注意，RemovalListener 抛出的任何异常都会在记录到日志后被丢弃[swallowed]。

5.5.3 Java 客户端使用

Guava 的 Maven 依赖包如下：

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>19.0</version>
</dependency>
```

本实例给读者演示了 GuavaCache 的两种使用方式：cacheLoader 和 callable callback。

```
package org.light.guava;

import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;
import com.google.common.cache.Cache;
import com.google.common.cache.CacheBuilder;
import com.google.common.cache.CacheLoader;
import com.google.common.cache.LoadingCache;
import com.google.common.collect.ImmutableMap;
import com.google.common.collect.Lists;

public class TestGuavaCache {
    private LoadingCache<String,String>cahceBuilder;

    public TestGuavaCache(){
        this.init();
    }
    //cacheLoader 初始化
    private void init(){
        CacheLoader<String, String>cacheLoader = new CacheLoader
<String, String>(){
            @Override //非命中 key 单条数据加载
            public String load(String key) throws Exception {
                String strProValue="hello load "+key+"!";
                return strProValue;
            }

            @Override
            public  Map<String,String>loadAll(Iterable<?  extends
String>iterable) throws Exception {//非命中 keys 批量加载
        Map<String,String>rtMap = new HashMap<>();
        Iterator<? extends String>iters = iterable.iterator();
        String tmpKey = null;
        while(iters.hasNext()){
            tmpKey = iters.next();
            rtMap.put(tmpKey, "hello loadAll "+tmpKey+" !");
        }
                return rtMap;
            }
        };
        cahceBuilder = CacheBuilder.newBuilder()
                //设置并发级别为 10，并发级别是指可以同时写缓存的线程数
                        .concurrencyLevel(10)
                //设置缓存最大容量为 10000，超过之后就会按照 LRU
                //最近虽少使用算法来移除缓存项
                        .maximumSize(10000)
                //根据某个键值对最后一次访问之后 5 分钟后移除
                    .expireAfterAccess(5, TimeUnit.MINUTES)
```

```
                // 根据某个键值对被创建或值被替换后 5 分钟移除
                        .expireAfterWrite(5, TimeUnit.MINUTES)
                // 给定时间：1 分钟内没有被读/写访问，则回收
                        .refreshAfterWrite(1, TimeUnit.MINUTES)
                            .build(cacheLoader);
    }
    //测试 cacheLoader 方式缓存基本操作
    public void testLoadingCache() throws Exception{
        System.out.println("jerry value : "
                        +cahceBuilder.get ("jerry"));//直接获取
        System.out.println("peida value : "
                            +cahceBuilder.get ("peida"));
        cahceBuilder.put("harry", "ssdded");//直接设置<k,v>缓存
        System.out.println("harry value : "
                            +cahceBuilder.get ("harry"));
        List<String>params = Lists
                            .newArrayList("灵魂", "歌唱", "内心");
        ImmutableMap<String,String> rts
                                = cahceBuilder.getAll(params);
        System.out.println(rts);
        cahceBuilder.invalidate("jerry");   //单个清除
        cahceBuilder.invalidateAll(params); //批量清楚
        cahceBuilder.invalidateAll();       //清楚所有缓存项
    }
    //callback 使用示例
    public void testcallableCache()throws Exception{
        Cache<String, String> cache = CacheBuilder. newBuilder()
                                        .maximumSize(1000).build(
                                    );
        String resultVal=cache.get("jerry",new Callable<String>(){
            public String call() {
                String strProValue="hello "+"jerry"+"!";
                return strProValue;
            }
        });
        System.out.println("jerry value : " + resultVal);

        resultVal = cache.get("peida", new Callable<String>() {
            public String call() {
                String strProValue="hello "+"peida"+"!";
                return strProValue;
            }
        });
        System.out.println("peida value : " + resultVal);
    }


    public void test(){
        //cacheLoader 方式使用示例
        try {
            testLoadingCache();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
        //callback 方式使用示例
        try {
            this.testcallableCache();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        TestGuavaCachetgc = new TestGuavaCache();
        tgc.test();
    }
}
```

这里只是对 GuavaCache 的简单使用示例，生产环境下实际开发时，随着业务需要，在现有示例基础上需要做相应调整。

5.6 总结

本章主要介绍网络内存数据数据库 Redis3.0 及 SSDB 和本地缓存 GuavaCache，首先介绍 Redis3.0 的新特性，Redis3.0 服务端集群架构、容错机制和 SSDB 基本认识，接着介绍 Redis3.0 基本配置及使用方式，并结合对比了号称 Redis 替代方案的 SSDB，接着向读者展示了 Redis 及 SSDB 的完整使用示例，让读者加深对 Redis 和 SSDB 的认识。最后向读者介绍精确控制本地缓存 GuavaCache 的原理及用法实例。

笔者建议：如果内存资源需求不是特别大，内存数据库可以考虑使用 SSDB；反之，建议大家使用 Reidis3.0 集群模式。如果内存数据库 IO 特别频繁，则考虑使用 GuavaCache 配合网络缓存。

NoSQL 在数据库领域的影响力日渐壮大，领头羊 MongoDB，Cassandra 和 Redis 已经在 DB-Engine 的数据库排名上进入了前十，HBase 也进入了前十五，所以许多的大数据架构、平台也都在越来越强化对于 NoSQL 数据库的支持。因此下章主要介绍基于键值对方式存储的文档数据库代表 Mongo3.0 和基于列簇方式存储的代表 HBase1.0。

第6章 NoSQL：MongoDB3.0和HBase1.0

MongoDB 是 2007 年起由 10gen 公司进行改良并在 2009 年成为拥有 AGPL 许可的开源项目。MongoDB 将自身定位为一个开源的、易于扩展的文档型数据库。每条记录实际上都是以一个文档存在于 MongoDB 中的，文档以 JSON 格式，二进制 JSON (BSN) 存储在 MongoDB 中，BSON 文档是包含所存元素有序列表的对象，每条元素都由一个字段名和一个特定类型的值组成。Mongo3.0 在性能、压缩及运维层面都有极大提升。

HBase——Hadoop Database，始于 Apache Hadoop 一个子项目，大约在 2007 年随同 Hadoop 一起发布，三年后，HBase 成为一个独立的 Apache 顶级项目。HBase 是一个高可靠性、高性能、面向列、可伸缩的开源分布式 NoSQL 数据库，它源于 Fay Chang 所撰写的 Google 论文“Bigtable（一个结构化数据的分布式存储系统）”的设计思想。HBase 构建在 Hadoop 基础设施之上，用户使用它能够在廉价 PC Server 上搭建起大规模结构化存储集群。经过了七年的研发，HBase1.0 版正式发布，这次发布提供了一些令人兴奋的特性和并未牺牲稳定性的新 API，且保持向后兼容。

6.1 MongoDB3.0 和 HBase1.0 新特性

当前 MongoDB 最新版是 3.2，HBase 最新版是 1.2，但可以说 MongoDB3.0 和 HBase1.0 均是各自发展史上的一个巨大的飞跃。

6.1.1 MongoDB3.0 新特性

MongoDB3.0 带来一系列让人惊喜的新特性[1]，下面为其几个核心点。

1. 插件式存储引擎API

MongoDB 3.0 引入了插件式存储引擎 API，为第三方的存储引擎厂商加入 MongoDB 提供了方便，这一变化无疑参考了 MySQL 的设计理念。目前除了早期的 MMAP 存储引擎外，WiredTiger 和 RocksDB 均已完成了对 MongoDB 的支持，前者更是在被 MongoDB 公司收购后直接引入 MongoDB 3.0 版本中，如图 6.1 所示，插件式存储引擎 API 的引入为 MongoDB 丰富自己武器库，以处理更多不同类型的业务提供了无限可能，内存存储引擎、事务存储引擎甚至 Hadoop 在未来都有可能接进来。

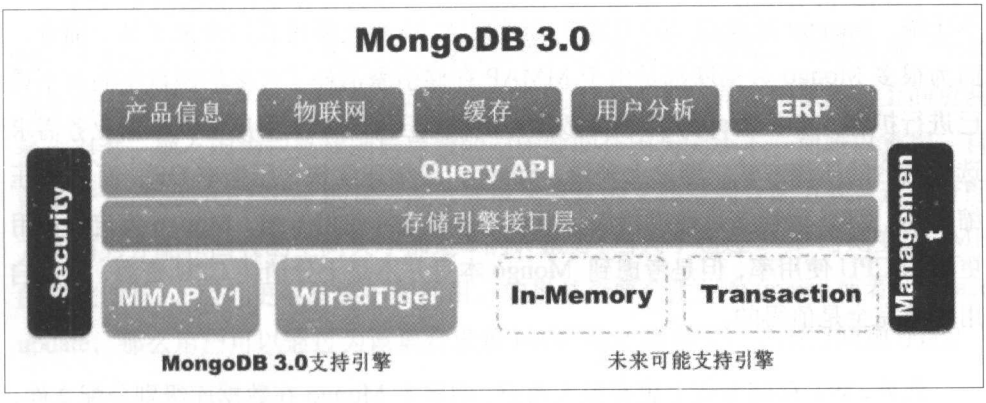


图 6.1 MongoDB 3.0 架构

1 MongoDB3.0 新增特性一览 http://blog.sina.com.cn/s/blog_48c95a190102vedr.html

2．WiredTiger存储引擎（WT）

如果说插件式存储引擎 API 为 MongoDB 3.0 打造了一个武器库，那么 WiredTiger 绝对是武器库中的第一枚也是最重要的一枚重磅炸弹。因为 MMAP 存储引擎自身的天然缺陷（耗费磁盘空间和内存空间且难以清理，库级别锁），MongoDB 为数据库运维人员带来了极大痛苦，甚至一部分人已经开始转向 TokuMX，尽管后者目前也不甚稳定。意识到这一问题的 MongoDB，做出了有钱任性的决定，直接收购存储引擎厂商 WT，将 WT 存储引擎集成进 3.0 版本（仅在 64 位版本中提供）。WT 存储引擎特性如下。

（1）文档级别并发控制：WT 通过 MVCC 实现文档级别的并发控制，即文档级别锁。它允许多个客户端请求同时更新一个集合内存的多个文档，而再也不需要在排队等待库级别的写锁。这在提升数据库读写性能的同时，大大提高了系统的并发处理能力。关于这一点的效果从监控工具 mongostat 可直接体现出来，旧版本的监控指标会有 locked db 这一项（该项指标过高是 mongo 使用人员的一大痛点），而新版的 mongostat 已经看不到了。

（2）磁盘数据压缩：WT 支持对所有集合和索引进行 Block 压缩和前缀压缩（如果数据库启用了 journal，journal 文件一样会压缩），已支持的压缩选项包括：不压缩、Snappy 压缩和 Zlib 压缩。这为广大 Mongo 使用者们带来了又一福音，因为很多 Mongo 数据库都是由于 MMAP 存储引擎消耗了过多的磁盘空间而不得已进行扩容。其中 Snappy 压缩为数据库的默认压缩方式，用户可以根据业务需求选择适合的压缩方式。理论上来说，Snappy 压缩速度快，压缩率 OK，而 Zlib 压缩率高，CPU 消耗多且速度稍慢。当然，只要选择使用压缩，Mongo 肯定会占用更多的 CPU 使用率，但是考虑到 Mongo 本身并不是十分消耗 CPU 资源，所以启用压缩完全是值得的。

此外，WT 存储方式上也有很大改进。旧版本 Mongo 在数据库级别分配文件，数据库中的所有集合和索引都混合存储在数据库文件中，所以即使删掉了某个集合或者索引，占用的磁盘空间也很难及时自动回收。WT 在集合和索引级别分配文件，数据库中的所有集合和索引均存储在单独的文件中，集合或者索引删除后，

对应的存储文件随即删除。当然，因为存储方式不同，低版本的数据库无法直接升级到 WT 存储引擎，只能通过导出导入数据的方式来实现。

（3）可配置内存使用上限：WT 支持内存使用容量配置，用户可以控制 MongoDB 所能使用的最大内存，只要修改 storage.wiredTiger.engineConfig.cacheSizeGB 参数即可，该参数默认值为物理内存大小的一半。这也为广大 Mongo 使用者们带来了又一福音，MMAP 存储引擎消耗内存是出了名的，只要数据量够大，简直就是有多少用多少。

3．MMAPv1存储引擎提升

原有的存储引擎 MMAP 也进行了一定的完善，该存储引擎依然是 3.0 版的默认存储引擎。遗憾的是改进后的 MMAP 存储引擎依旧在数据库级别分配文件，数据库中的所有集合和索引都混合存储在数据库文件中，所以磁盘空间无法及时自动回收的问题如故。

（1）锁粒度由库级别锁提升为集合级别锁：在一定程度上也能够提升数据库的并发处理能力。

（2）文档空间分配方式改变：MongoDB 3.0 版本中的 MMAPv1 抛弃了基于 paddingFactor 的自适应分配方式，因为这种方式看起来很智能，但是因为一个集合中的文档大小不一，所以经过填充后的空间大小也不一样。如果集合上的更新操作很多，那么因为记录移动后导致的空闲空间会因为大小不一而难以重用。目前基于 usePowerOf2Sizes 的预分配方式成为默认的文档空间分配方式，这种分配方式因为分配和回收的空间大小都是 2 的 N 次方（当大小超过 2MB 时则变为 2MB 的倍数增长），因此更容易维护和利用。如果某个集合上只有 insert 或者 in-place update，那么用户可以通过为该集合设置 noPadding 标志位，关闭空间预分配。

4．复制集改进

（1）**复制集成员增长：**MongoDB 3.0 的复制集成员的最大个数由之前的 12 个增长为 50 个，但能够投票的最大成员个数依然为 7 个，而相应的 getLastError

中的 w:“majority”项也仅代表投票节点的大多数。

(2)**Primary 节点 StepDown 处理方式变化**：在复制集中通过 replSetStepDown 命令可以使得当前的 Primary 节点退位，重新选举新的 Primay 节点。MongoDB 3.0 在 StepDown 的处理方式上做了如下修改：

- 在 Primary 退位之前，会首先中断某些耗时较长的用户操作如创建索引、写操作、Mapreduce 任务等；
- 为了防止数据回滚，Primary 节点在退位之前会等待一个可被选举的 Secondary 节点同步到最新数据，而旧版本中 Primary 节点只要有 Secondary 节点的数据同步到 10 秒以内就退位；
- replSetStepDown 命令新增了一个 secondaryCatchUpPeriodSecs 参数，用户可以指定 Primary 节点等待有 Secondary 节点的数据同步到该参数指定的秒数内就退位。

5．分片集群改进

(1) 新增工具函数 sh.removeTagRange()：旧版本中只有 sh.addTagRange()，如果要删除 tagRange 只能手工到 config.tags 集合中删除。

(2) 提供更可预测的 Read Preference 处理：新版本中 mongos 实例在执行读操作时不再将连接固定在复制集成员上，而是对每个读操作都会重新评估 Read Preference。这样当 Read Preference 修改时，其行为更容易预测。

(3) 为 chunk 迁移提供 writeConcern 设置：新版本针对均衡器为 moveChunk 和 cleanupOrphaned 这两个涉及 chunk 迁移的命令提供了 writeConcern 参数。

(4) 增加均衡器状态显示：新版本中通过 sh.status()可以看到均衡器的状态信息。

6.1.2 HBase1.0 新特性

当前，依托于 Hadoop 的迅猛发展，HBase 在大数据领域的应用越累越广。

Apache HBase社区发布了Apache HBase 1.0.0。它花费了七年时间在Apache HBase项目领域取得了里程碑式的发展[2]，1.0.0 版本有三个目标：

（1）为将来的 1.x 系列版本奠定稳定基础。

（2）稳定运行的 HBase 集群及客户端。

（3）让版本和兼容性方面更加明确。

包括之前的 0.99.x 系列版本，1.0.0 解决了超过 1500 个 JIRA 跟踪的问题。该版本值得关注的改进包括：

（1）性能提升，在保持之前的稳定性的情况下，实现了性能的提升。

（2）新增了 API 和对客户端 API 进行了重组和改变：1.0.0 引进了新的 API，并且废弃了一些常用的客户端 API（HTableInterface, HTable 和 HBaseAdmin）。笔者建议新的应用程序来使用新风格的 API，因为这些废弃的 API 在将来 2.x 系列版本之后被删除。详细信息请参考：

```
http://www.slideshare.net/xefyr/apache-hbase-10-release
http://s.apache.org/hbase-1.0-api。
```

（3）使用时间轴一致区域副本以达到新的可用性保证：一个区域可以以只读模式放在多个区域服务器上。该区域副本之一将会是主服务器，支持写入，其他副本将共享与之相同的数据。对复制副本的读请求可以为后备的 RPC 请求来提供时间连续的高可用性。

（4）联机配置进行了改进，从而在无需重启区域服务器的情况下，就能够重新加载服务器配置的子集。

（5）完善了相关文档，增强了可使用性。优化了很多性能（优化 WAL 管道、disruptor 的使用，多 WAL 以及更多使用 off-heap 内存）。

---

2 七年磨一剑：HBase1.0 正式发布 http://www.infoq.com/cn/news/2015/03/apache-hbase-1-release

6.1.3 MongoDB和HBase比较

二者皆为当前主流NoSQL，其区别[3]如下：

（1）MongoDBbson文档型数据库，整个数据都存在磁盘中；HBase是列式数据库，依赖于HDFS，集群部署时每个familycolumn保存在单独的hdfs文件中。

（2）MongoDB支持二级索引，而HBase本身不支持二级索引。

（3）HBase一个region只有一个HRegionServer对外提供服务（没有负载均衡的概念）；MongoDB的shards（类似于region）支持负载均衡。HBase根据文件的大小来控制region的分裂；MongoDB根据负载来决定shards的分裂。

（4）Mongodb主键是“_id”，主键上面可以不建索引，记录插入的顺序和存放的顺序一样，HBase的主键就是row key，可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在HBase内部，row key保存为字节数组。存储时，数据按照Row key的字典序(byte order)排序存储。设计key时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。

（5）MongoDB支持集合查找、正则查找、范围查找，支持skip和limit等等，是最像MySQL的NoSQL数据库，而HBase只支持三种查找方式：通过单个row key访问，通过row key的range，全表扫描。

（6）MongoDB的update是update-in-place，也就是原地更新，除非原地容纳不下更新后的数据记录。而HBase的修改和添加都是同一个命令：put，HBase内部也不是更新，它只是将这一份数据以不同的版本保存下来而已，HBase默认的保存版本的历史数量是3。

（7）MongoDB和HBase都支持MapReduce，不过MongoDB的MapReduce支持不够强大，如果没有用MongoDB分片，MapReduce实际上不是并行执行的。

（8）MongoDB支持shard分片，HBase根据row key自动负载均衡，这里shard

3 MongoDB和Hbase特点分析 http://blog.itpub.net/24288/viewspace-1120421/

key 和 row key 的选取尽量用非递增的字段，尽量用分布均衡的字段。

(9) MongoDB 的读效率比写高，HBase 默认适合写多读少的情况，可以通过 hfile.block.cache.size 配置，该配置 storefile 的读缓存占用 Heap 的大小百分比，该值直接影响数据读的性能。设置此值的时候，你要同时参考 hbase.regionserver.global.memstore.upperLimit，该值是 memstore 占 heap 的最大百分比，两个参数一个影响读，一个影响写。如果两值加起来超过 80%~90%，会有 OOM 的风险，需要谨慎设置。

(10) HBase 采用的 LSM 思想 (Log-Structured Merge-Tree)，就是将对数据的更改 hold 在内存中，达到指定的 Threadhold 后将该批更改 merge 后批量写入到磁盘，这样将单个写变成了批量写，大大提高了写入速度，不过这样的话读的时候就费劲了，需要 merge disk 上的数据和 memory 中的修改数据，这显然降低了读的性能。MongoDB 采用的是 mapfile+Journal 思想，如果记录不在内存，先加载到内存，然后在内存中更改后记录日志，隔一段时间批量写入 data 文件，这样对内存的要求较高，至少需要容纳下热点数据和索引。

6.2 MongoDB3.0 集群和索引

6.2.1 MongoDB3.0 集群

MongoDB 有三种集群方式：Sharding、Replica Set 和 Master / Slaver。其中 Sharding 是在集合层面做数据分片，以 shard key 来分片，即把一个表的数据分散存储在多个 MongoDB 服务节点上。后两者是 MongoDB 所支持的两种复制模式：

Master/Slave，主从复制，角色包括 master 和 slave。

Replica Set，复制集复制，角色包括 primary 和 secondary。

相比 master-slave，replica set 的优点就是没有单点故障，primary 故障之后，整个 replica set 会自动选择一个健康的节点成为 primary，承担写的任务，可用性比 master-slave 的高，提供更高的可用性。

MongoDB 官方给的集群架构如图 6.2 所示，从中了解到整个集群主要有 4 个模块[4]：Config Server、mongos、shard、replica set。

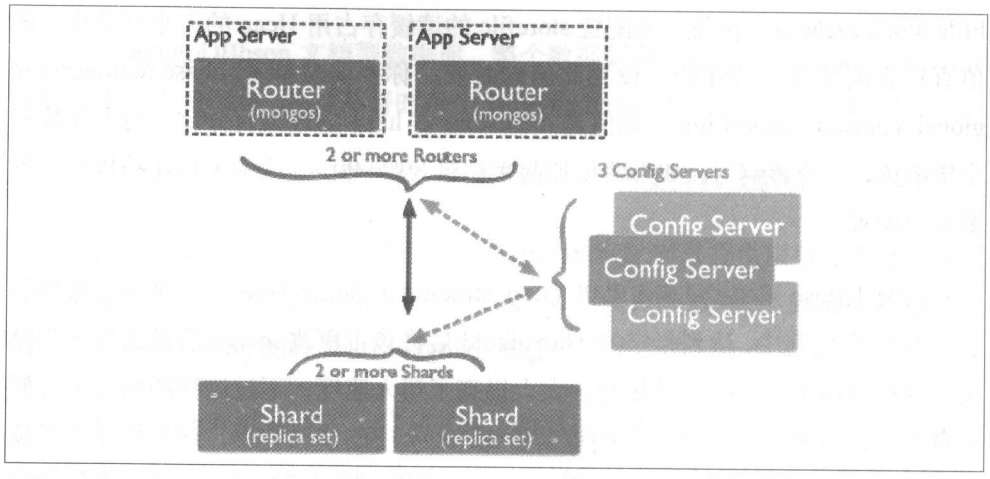


图 6.2 Mongo3.0 集群架构

Config Server：用来存放集群的元数据，也就是存放所有分片的配置数据，mongos 第一次启动就需要连接 configServer 读取相关数据，当 configServer 有数据进行更新时，也会主动推送消息到所有的 mongos 上，在 3.0.7 版本中，官方是建议配置 3 份的 Config Server，以便挂掉两台时，业务还能够正常运转。

mongos：查询路由，负责 client 的连接，并把任务分发给 shards，然后收集结果。Mongodb 集群的请求入口，能否自动实现数据的分布式分发，生产环境中建议部署在应用服务器上。

shard：即数据结点，存储数据和执行计算。分片就比如是将一张大表分散在几个不同的 shard 中，实现数据分布式存储。为了保证高可用和数据一致性，生产环境中 shards 应该做成 replicasets(防止丢失数据)。集群中有一个 primary shards，执行非分片的任务。

replica set：主要是对每个分片进行冗余，生产环境中，一般将副本集配置在

4 MongoDB3.2 集群搭建 http://www.linuxidc.com/Linux/2016-01/127437.htm

三个节点上，两份副本、一份仲裁。

6.2.2 Mongo 索引介绍

新版 Mongo3.0 后台索引建立过程中，不能进行删库、删表、删索引的操作，且后台索引建立的过程不会因此自动中断。另外，使用 createIndexes 命令可以同时建立多个索引，并且只扫描一遍数据，提升了建索引的效率。

工作中用得比较多的索引，除了默认主键索引，还有稀疏索引及 TTL 索引等。“$”符号不可以作为索引的首字母，“.”不能在索引名的任何位置出现。

1. 基础索引

建立索引的函数 ensureIndex()，在表 person 的 name 列上建立索引：1（升序）、-1（降序），默认为升序，例如：db.person.ensureIndex({ name : 1 })。

_id 是创建表的时候自动创建的索引，此索引是不能被删除的。

当系统已有大量数据时，创建索引就是个非常耗时的活，我们可以在后台执行，只需指定“backgroud:true”即可，例如：db.person.ensureIndex({age:1} , {backgroud:true})。

2. 索引模式集创建方式

（1）创建唯一索引：只需要在 ensureIndex 命令中指定"unique:true"，即可创建唯一索引。例如：db.person.ensureIndex({age:1} , {unique:true});创建了唯一索引的字段不允许插入相同的值。

（2）创建组合索引：当创建组合索引时，字段后面的 1 表示升序，-1 表示降序，是用 1 还是用-1，主要跟排序的时候或指定范围内查询的时候有关。

例如：db.persion.ensureIndex({name:1 , age:1})。

（3）创建稀疏索引：如果数据里的一些行中没有某个字段或字段值为 Null，

那么在这个字段上建立普通索引时，无此字段或值Null的行也会参与到索引结构中，占用相应的空间。如果不希望这些值为空的行参与到索引中，这时候可以采用松散索引，松散索引只会让指定字段不为空的行参与到索引创建中来。创建一个松散索引可以用下面的命令：db.person.ensureIndex({user_id: 1}, {sparse: true})。

（4）创建多值索引：即可以对一个array类型创建索引，比如像下面的结构：{ name: "Wheelbarrow",tags: ["tools", "gardening", "soil"]}，MongoDB可以在tags字段上创建索引：

```
db.person.ensureIndex({tags: 1})
```

在生成索引时，会为tags中的三个值分别生成三个索引元素，索引中tools，gardening，soil三个值都会指向这同一行数据，相当于分裂成了三个独立的索引项。

（5）创建TTL索引（Time-To-Liveindex）：TTL索引允许为每一个文档设置一个超时时间，超过这个时间文档就会被删除。在ensureIndex中指定expireAfterSecs选项就可以创建一个TTL索引。例如：db.userShortTags.ensureIndex ({"adt": 1}, {expireAfterSeconds: 3600 * 8})

上述命令对用户短期兴趣标签设置TTL索引为8小时。

其他常用索引类型，还有空间索引、全文索引和嵌套索引，详细信息可以参考官网。

6.3 HBase底层实现介绍

6.3.1 HBase相关Hadoop体系

HBase是Google Bigtable的开源实现，类似Google Bigtable利用GFS作为其文件存储系统，HBase利用Hadoop HDFS作为其文件存储系统；Google运行MapReduce来处理Bigtable中的海量数据，HBase同样利用HadoopMapReduce来处理HBase中的海量数据；Google Bigtable利用Chubby作为协同服务，HBase

利用 Zookeeper 作为对应。

图 6.3 描述了 Hadoop EcoSystem 中的各层系统[5]，其中 HBase 位于结构化存储层，Hadoop HDFS 为 HBase 提供高可靠性的底层存储支持，HadoopMapReduce 为 HBase 提供高性能的计算能力，Zookeeper 为 HBase 提供稳定服务和 failover 机制。Pig 和 Hive 还为 HBase 提供了高层语言支持，使得在 HBase 上进行数据统计处理变得非常简单。Sqoop 则为 HBase 提供了方便的 RDBMS 数据导入功能，使得传统数据库数据向 HBase 中迁移变得非常方便。

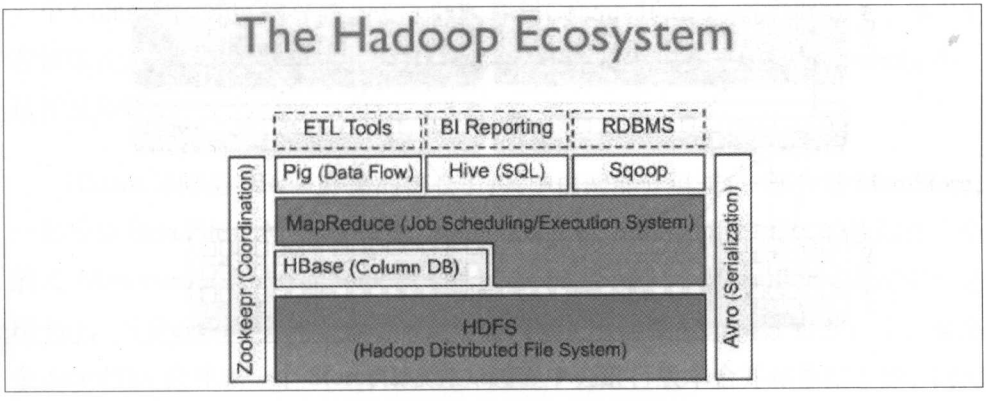


图 6.3 Hadoop Ecosystem

当 Table 随着记录数不断增加而变大后，会逐渐分裂成多份 splits，成为 regions，一个 region 由[startkey,endkey]表示，不同的 region 会被 Master 分配给相应的 RegionServer 进行管理。HBase 中有两张特殊的 Table，.META.和-ROOT-：

- .META.：记录了用户表的 Region 信息，.META.可以有多个 region。
- -ROOT-：记录了.META.表的 Region 信息，-ROOT-只有一个 region。

Zookeeper 中记录了-ROOT-表的 location，Client 访问用户数据之前需要首先访问 Zookeeper，然后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的位置去访问，中间需要多次网络操作，不过 client 端会做 cache 缓存。

5 HBase 原理、基本概念、基本结构 http://blog.csdn.net/woshiwanxin102213/article/details/17584043

6.3.2 HBase 系统架构

如图 6.4 所示，HBase 整个系统架构主要由 Client、ZooKeeper、HMaster、HReginServer 及其存储组成。

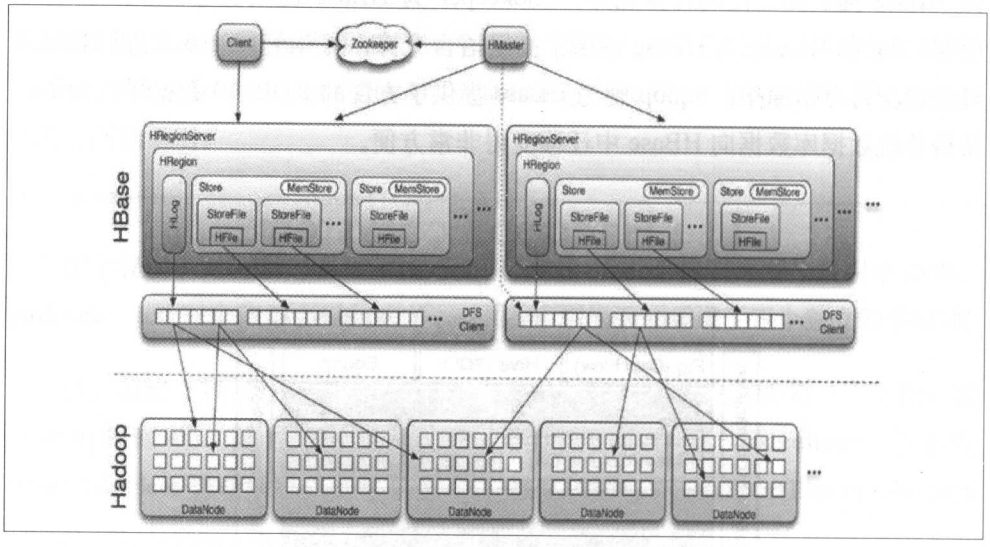


图 6.4 HBase 系统架构

（1）Client：使用 HBase 的 RPC 机制与 HMaster 和 HRegionServer 进行通信。对于管理类操作，Client 与 HMaster 进行 RPC；对于数据读写类操作，Client 与 HRegionServer 进行 RPC。

（2）Zookeeper：Zookeeper Quorum 中除了存储了-ROOT-表的地址和 HMaster 的地址，HRegionServer 也会把自己以 Ephemeral 方式注册到 Zookeeper 中，使得 HMaster 可以随时感知到各个 HRegionServer 的健康状态。此外，Zookeeper 也避免了 HMaster 的单点问题。

（3）HMaster：没有单点问题，HBase 中可以启动多个 HMaster，通过 Zookeeper 的 Master Election 机制保证总有一个 Master 在运行，它主要负责 Table 和 Region 的管理工作：

- 管理用户对 Table 的增删改查操作。

- 管理 HRegionServer 的负载均衡，调整 Region 分布。
- 在 Region Split 后，负责新 Region 的分配。
- 在 HRegionServer 停机后，负责失效 HRegionServer 上 Region 迁移。

（4）HRegionServer：主要负责响应用户 I/O 请求，向 HDFS 文件系统中读写数据，是 HBase 中最核心的模块。

HRegionServer 内部管理了一系列 HRegion 对象，每个 HRegion 对应了 Table 中的一个 Region，HRegion 中由多个 HStore 组成。每个 HStore 对应了 Table 中的一个 Column Family 的存储，可以看出每个 Column Family 其实就是一个集中的存储单元，因此最好将具备共同 IO 特性的 column 放在一个 Column Family 中，这样最高效。

HStore 存储是 HBase 存储的核心了，其中由两部分组成，一部分是 MemStore，一部分是 StoreFiles。MemStore 是 Sorted Memory Buffer，用户写入的数据首先会放入 MemStore，当 MemStore 满了以后会 Flush 成一个 StoreFile（底层实现是 HFile），当 StoreFile 文件数量增长到一定阈值，会触发 Compact 合并操作，将多个 StoreFiles 合并成一个 StoreFile，合并过程中会进行版本合并和数据删除，因此可以看出 HBase 其实只有增加数据，所有的更新和删除操作都是在后续的 compact 过程中进行的，这使得用户的写操作只要进入内存中就可以立即返回，保证了 HBase I/O 的高性能。当 StoreFiles Compact 后，会逐步形成越来越大的 StoreFile，当单个 StoreFile 大小超过一定阈值后，会触发 Split 操作，同时把当前 Region Split 成 2 个 Region，父 Region 会下线，新 Split 出的 2 个孩子 Region 会被 HMaster 分配到相应的 HRegionServer 上，使得原先 1 个 Region 的压力得以分流到 2 个 Region 上。

每个 HRegionServer 中都有一个 HLog 对象，HLog 是一个实现 Write Ahead Log 的类，在每次用户操作写入 MemStore 的同时，也会写一份数据到 HLog 文件中，HLog 文件定期会滚动出新的，并删除旧的文件（已持久化到 StoreFile 中的数据）。当 HRegionServer 意外终止后，HMaster 会通过 Zookeeper 感知到，HMaster 首先会处理遗留的 HLog 文件，将其中不同 Region 的 Log 数据进行拆分，分别放到相应 region 的目录下，再将失效的 region 重新分配，领取到这些 region 的

HRegionServer 在 Load Region 的过程中，会发现有历史 HLog 需要处理，因此会“Replay” HLog 中的数据到 MemStore 中，然后“flush”到 StoreFiles，完成数据恢复。

5．HBase容错性

（1）Master 容错：Zookeeper 重新选择一个新的 Master，无 Master 过程中，数据读取仍照常进行；无 master 过程中，region 切分、负载均衡等无法进行。

（2）RegionServer 容错：定时向 Zookeeper 汇报心跳，如果一旦时间内未出现心跳，Master 将该 RegionServer 上的 Region 重新分配到其他 RegionServer 上，失效服务器上“预写”日志由主服务器进行分割并派送给新的 RegionServer。

（3）Zookeeper 容错：Zookeeper 是一个可靠的服务，一般配置 3 或 5 个 Zookeeper 实例。

6．HBase预分区Pre－splitting

当一个 table 刚被创建的时候，HBase 默认给 table 分配一个 region。也就是说这个时候，所有的读写请求都会访问到同一个 regionServer 的同一个 region 中，这个时候就达不到负载均衡的效果了，集群中的其他 regionServer 就可能会处于比较空闲的状态。解决这个问题可以用 pre-splitting，在创建 table 的时候就配置好，生成多个 region。

在 table 初始化的时候如果不配置的话，HBase 是不知道如何去 split region 的，因为 HBase 不知道应该哪个 row key 可以作为 split 的开始点。如果可以大概预测到 row key 的分布，则可以使用 pre-spliting 来帮助我们提前 split region。

不过如果我们预测得不准确的话，还是可能导致某个 region 过热，被集中访问，不过还好我们还有 auto-split。最好的办法就是首先预测 split 的切分点，做 pre-splitting，然后让 auto-split 来处理后面的负载均衡。

HBase 自带了两种 pre-split 的算法，分别是 HexStringSplit 和 UniformSplit。

如果 row key 是十六进制的字符串作为前缀的，就比较适合用 HexStringSplit 作为 pre-split 的算法；如果表查询只是以随机查询为主，则可以用 UniformSplit 方式进行，它是按照原始 byte 值（从 0x00~0xFF）右边以 00 填充。以这种方式分区的表在插入的时候需要对 rowkey 进行一个技巧性的改造，比如原来的 rowkey 为 rawStr，则需要对其取 hashCode，然后进行按照比特位反转后放在最初 rowkey 串的前面。

这里我们给出一个创建表时预分区的 HBasel Shell：

```
    hbase(main):028:0> create 'user_features', {NUMREGIONS => 128,
SPLITALGO => 'UniformSplit'},{ NAME => 'f', COMPRESSION => 'SNAPPY',
IN_MEMORY => 'true', BLOCKCACHE => 'true' , TTL => 259200}
    0 row(s) in 55.0570 seconds
    =>Hbase::Table - user_features
```

该建表语句，创建一个预设的 128 个分区，分区算法采用 UniformSplit，含有一个列簇“f”，对内容采用“SNAPPY”算法，设置 TTL 时间为 3 天。

6.4 Mongo 和 HBase 客户端使用

6.4.1 Mongo 客户端

官方所提供 MongoDB Java Driver 来自 http://mongodb.github.io/mongo-java-driver/。目前 MongoDB 最新版 3.2.2，其最新版 Maven 依赖如下所示：

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.2.2</version>
</dependency>
```

这里向读者介绍 Mongo3.0 连接注意事项，这里应该把 Mongo 集群中相关节点都加到 Mongo 的 Connection 中。运行过程中，一旦有节点变更，Mongo 主从节点调整，系统也会自动调整。下面给出一个 MongoDB 基本操作的完整实例，代码如下：

```
import java.util.ArrayList;
```

```
import java.util.Calendar;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.bson.Document;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.MongoClient;
import com.mongodb.MongoClientOptions;
import com.mongodb.MongoCredential;
import com.mongodb.ReadPreference;
import com.mongodb.ServerAddress;
import com.mongodb.WriteConcern;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;
import org.light.rtc.util.Constants;

public class MongoDao {
    private static MongoDao instance;
    private MongoClientmongoClient;
    private MongoDatabasemgdb;
    private Document sortDoc = new Document();

    private void init(){
        List<ServerAddress>addrList      =      new      ArrayList
<ServerAddress>();
        //解析实现配置好的Mongo hosts字符串，加入所有节点
        String[] mgHost = Constants.mongoHost.split(",");
        for(String hst : mgHost){
            addrList.add(new ServerAddress(hst,
                              Constants. mongoPort));
        }
        //安全模式认证
        MongoCredential credentials = MongoCredential
                          .createScramSha1Credential( "light2016",
                          "user_logs", "admin2016".toCharArray());
        List<MongoCredential>credentialsList
                               = new ArrayList<MongoCredential>();
        credentialsList.add(credentials);
        //客户端连接参数设定
        MongoClientOptions.Builder builder
                                  = MongoClientOptions. builder();
        builder.connectionsPerHost(50);
      builder.threadsAllowedToBlockForConnection Multiplier(200);
        builder.maxWaitTime(1000*60*10);
        builder.writeConcern(WriteConcern.REPLICA_ACKNOWLEDGED);
     builder.readPreference(ReadPreference.secondary Preferred());
        builder.connectTimeout(1000*60*1);
```

```
        MongoClientOptionsmco = builder.build();
        //获取 mongo 客户端
        mongoClient = new MongoClient(addrList,
                                      credentialsList, mco);
        //获取预操作的数据库
        mgdb = mongoClient.getDatabase(Constants.mongoDb);
        sortDoc.put("adt", -1);
    }

    private MongoDao(){
        this.init();
    }

    //单例模式初始化
    public synchronized static MongoDaogetInstance(){
        if(instance==null){
            instance = new MongoDao();
        }
        return instance;
    }

    //批量删除指定时间范围段日志
    public void delBatchLogs(){
        Document queryDoc = new Document();
        Document timeDoc = new Document();
        timeDoc.put("$gte", 201605032020);
        timeDoc.put("$lt", 201605032030);
        queryDoc.append("adt", timeDoc);
        //删除指定时间段的日志并返回删除的日志条数
        int delNum = (int)this.mgdb.getCollection(
                        Constants. mongoVisitLogs)
                        .deleteMany(queryDoc).getDeletedCount();
        System.out.println(" 过去十分钟用户行为删除个数: "+delNum);
    }

    //批量插入访问日志
    public void addVisitLogs(List<Document>rdList){
        this.mgdb.getCollection("userLogs").insertMany(rdList);
    }

    /**
     * doc: _id, devId, lbs({lbId:relv}),adt
     * @paramdevId
     * @return 指定用户群中每个用户的短期兴趣标签及标签权重
     */
    public  Map<String,Map<String,Double>>  getUserShortTagsById
(Set<String>devIds){
        Map<String,Map<String,Double>>rtMap = null;
        Map<String,Double>tmpMap = null;
        MongoCursor<Document>mgCur = null;
        Document queryDoc = new Document(),subDoc=null,rtDoc = null;
        subDoc = new Document();
```

```
            subDoc.put("$in", devIds);
            //条件过滤：过滤出给定设备ID集合中的用户
            queryDoc.put("devId", subDoc);
            FindIterable<Document> filters = this.mgdb.getCollection
("userShortTags")
                                        .find(queryDoc).sort(sortDoc);
            if(filters!=null){
                rtMap = new HashMap<String,Map<String,Double>>();
                mgCur = filters.iterator();
                String devId = null;
                while(mgCur.hasNext()){
                    rtDoc = mgCur.next();
                    devId = rtDoc.getString("devId");
                    if(!rtMap.containsKey(devId)){
                    //返回之前插入数据时的相同<k,v>类型的Map对象，并转换
                        tmpMap = (Map<String,Double>)rtDoc.get("lbs");
                        rtMap.put(devId, tmpMap);
                    }
                }
                mgCur.close();
                mgCur = null;
            }
            return rtMap;
        }
        public void test(){
            this.delBatchLogs();
        }

        public static void main(String[] args) {
            MongoDaomgDao = new MongoDao();
            mgDao.test();
        }
    }
```

官方除了提供上述驱动外，还提供了异步调用驱动：**MongoDBAsync Driver**，其异步调用模式，在有很多并发执行任务情况下，可以更加充分利用服务器资源，使用它只需要在工程下引入：

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-async</artifactId>
    <version>3.2.2 </version>
</dependency>
```

其使用方式和前述有很大不同，下面给出一个测试实例代码：

```
import java.util.List;
import java.util.concurrent.CountDownLatch;
import org.bson.Document;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import com.google.common.collect.Lists;
import com.mongodb.ReadPreference;
import com.mongodb.ServerAddress;
import com.mongodb.WriteConcern;
import com.mongodb.async.SingleResultCallback;
import com.mongodb.async.client.MongoClient;
import com.mongodb.async.client.MongoClientSettings;
import com.mongodb.async.client.MongoClients;
import com.mongodb.async.client.MongoDatabase;
import com.mongodb.connection.ClusterSettings;
import com.mongodb.connection.ClusterType;
import com.typesafe.config.Config; //非常方便实用的配置文件加工工具
import com.typesafe.config.ConfigFactory;

public class DocumentDao {
    //slf4j 日志记录器
    private  static final Logger LOG
                = LoggerFactory.getLogger (DocumentDao.class);
    //加载默认配置文件，得到相应 Mongo 配置对象
    private static Config dmgConf = ConfigFactory.load()
                .getConfig("rootConf"). getConfig("mongoConf");
    //异步 MongoClient 为接口，同步 MongoClient 为类，二者在不同包结构下
    private MongoClient rs1Client;
    //异步 MongoDataBase 和同步 MongoDataBase 为不同包下接口
    private MongoDatabasedocDB;
    private String docTableName;

    public DocumentDao(){
        this.init();
    }

    private void init(){
        //加载 Mongo 集群节点
        List<ServerAddress> rs1Hosts = Lists.newArrayList();
        for(String host : dmgConf.getString("rs1-hosts")
            . split(",")){
            rs1Hosts.add(new ServerAddress(host));
        }
        //集群属性设置
        ClusterSettingsclusterSettings = ClusterSettings.builder()
                .hosts(rs1Hosts)
                .requiredClusterType (ClusterType.REPLICA_SET)
                .build();
        //客户端属性设置
        MongoClientSettings settings = MongoClientSettings
                        .builder()
                        .clusterSettings(clusterSettings)
            .writeConcern(WriteConcern. REPLICA_ACKNOWLEDGED)
            .readPreference(ReadPreference. secondaryPreferred())
            .build();
        rs1Client = MongoClients.create(settings);
```

```
        docDB = rs1Client.getDatabase(dmgConf.getString("rs1-db"));
        docTableName = dmgConf.getString("rs1-doc");
    }

    //实例：查询总数及返回第一条文档记录
    public void countDocNum(){
        //声明同步倒数计数器
        final CountDownLatch latch = new CountDownLatch(2);
        //查询指定表总数，返回查询结构时打印总数
        this.docDB.getCollection(this.docTableName).count(
                        (final Long count, final Throwable t)-> {
                            LOG.info("新闻个数："+count);
                            latch.countDown();
                        });
        //返回第一条记录时，直接打印，下面给出两种实现方法
        //方法1：显示声明，传统调用
        SingleResultCallback<Document>printDocument
                        = new SingleResultCallback <Document>() {
            @Override
            public void onResult(final Document document,
                                 final Throwable t) {
                System.out.println(document.toJson());
                latch.countDown();
            }
        };
        this.docDB.getCollection(this.docTableName)
                                .find().first(printDocument);

        //方法2：利用Java8函数式编程调用更简单
        this.docDB.getCollection(this.docTableName).find()
                .first(
                (final Document document, final Throwable t) ->{
                        System.out.println(document.toJson());
                        latch.countDown();
                });
        //同步计数器等待完成任务
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void test(){
        this.countDocNum();
    }

    public static void main(String[] args) {
        DocumentDao dd = new DocumentDao();
        dd.test();
    }
}
```

上述代码之所以添加同步倒数计数器，目的是为了避免异步调用在 MongoDB 没有返回结果前，主线程退出，导致不能正常返回结果，反而出现“MongoDB: No server chosen by ReadPreferenceServerSelector”异常。

此外，需要注意的是，笔者在实践中发现 Mongo 插入数据时，如果嵌套结构的二级 key 对应 value 值为 null，Mongo 中插不进去，且会报错。

MongoDB3.0 默认引擎还是 MMAP，因此使用前必须加上 wiredtiger 参数。其主要参数包括：

- 缓存大小——wiredTigerCacheSizeGB：默认物理内存的一半。
- 落盘间隔——syncdelay，默认一分钟。
- 压缩方式——wiredTigerCollectionBlockCompressor：snappy 或 zlib。

生产环境下，根据实际需要，做相应调整。

6.4.2 HBase 客户端

表 6-1 HBase shell 常用命令

名　称	命令表达式
创建表	create '表名称', '列名称 1','列名称 2','列名称 N'
添加 / 更新记录	put '表名称', '行名称', '列名称:', '值'
查看记录	get '表名称', '行名称'
查看表中记录总数	count '表名称'
删除记录	delete '表名','行名称' , '列名称'
删除一张表	第一步 disable '表名称' 第二步 drop '表名称'
查看所有记录	scan "表名称"
查看表某列中所有数据	scan "表名称" , ['列名称:']
查看集群状态	status
列出全部表	List
查看表描述	describe ‘表名称’
清空 / 重新创建表	truncate ‘表名’

如上表 6-1 所示为 HBase shell 常用命令，利用这些基本命令，可以完成表的创建、删除、插入和修改等基本功能。

目前HBase最新版1.2.1，其对应的Maven依赖如下：

```
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-client</artifactId>
    <version>1.2.1</version>
</dependency>
```

HBase shell常用命令如表6-1所示。

下面给出HBase在Java客户端下的基本操作实例。

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map.Entry;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.CellUtil;
import org.apache.hadoop.hbase.ClusterStatus;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.client.Table;
import org.apache.hadoop.hbase.util.Bytes;

public class HbaseDao {
    private Connection conn;
    private Configuration conf;
    private TableNametbName;

    public HbaseDao(){
        this.init();
    }

    private void init(){
        //实例化表名
        tbName = TableName.valueOf("user_tags");
        conf = HBaseConfiguration.create();
```

```
    //设置 zookeeper hosts
    conf.set("hbase.zookeeper.quorum",
                "192.168.10.1,192.168.10.2,192.168.10.3");
    conf.set("hbase.zookeeper.property.clientPort", "2181");
    try {
        conn  = ConnectionFactory.createConnection(conf);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
//获取 HBase 客户端连接
public Connection getConnection(){
    Connection conn = null;
    try {
        conn  = ConnectionFactory.createConnection(conf);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return conn;
}

//关闭指定客户端连接
public void closeConnection(Connection conn){
    if(!conn.isClosed()){
        try {
            conn.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

//创建 HBase 表实例
public void createTable(){
    Admin admin = null;
    try {
        admin = conn.getAdmin();
        //检验 HBase 中是否存在预创建的表
        if(!admin.tableExists(tbName)){//如果不存在
            HTableDescriptor desc
                            = new HtableDescriptor (tbName);
            desc.addFamily(new HColumnDescriptor("feautres"));
            hcd.setTimeToLive(259200);//三天过期时间
            hcd.setInMemory(true);//使用内存模式
            //设定压缩算法
            hcd.setCompressionType(Algorithm.SNAPPY);
            hcd.setCompressTags(true);//压缩设置
            //创建指定列簇的表
            admin.createTable(desc);
        }
    } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
    //根据 rowKey 返回数据
    public void getData(String rowKey){
        Table table = null;
        try {
            table = conn.getTable(tbName);
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        //构建请求对象
        Get get = new Get(Bytes.toBytes(rowKey));
        // 结果实例
        Result result = null;
        try {
            //执行查询
            result = table.get(get);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 单元格集合
        List<Cell>listCells = result.listCells();
        if(listCells!=null){
        int rowNum = 0;
        // 遍历单元格
        for (Cell cell : listCells) {
            System.out.println("列 "+(++rowNum)+" 列族:列名--"
            + Bytes.toString(CellUtil. cloneFamily(cell)) + " : "
        + Bytes.toString(CellUtil. cloneQualifier(cell))+" 列值: "
        + Bytes.toString(CellUtil. cloneValue(cell)));
            System.out.println("时间戳: " + cell.getTimestamp());
        }
       }
    }

    //扫描全表打印，一半仅用于测试，该操作非常耗时
    //如果真的执行一般都需要增加 rowkey 过滤
    public void testScan() throws IOException {
        Table table = conn.getTable(tbName);
        // 扫描器：针对全表的查询器
        ResultScannerresultScanner = table.getScanner(new Scan());
        // 结果迭代器
        Iterator<Result> results = resultScanner.iterator();
        while(results.hasNext()) {
            //Result result = (Result) results.next();
            Result result = results.next();
            List<Cell> cells = result.listCells();
            for(Cell cell : cells) {
                System.out.println("列族: "
                + Bytes.toString (CellUtil.cloneFamily(cell)));
```

```
                System.out.println("列名："
            + Bytes.toString (CellUtil.cloneQualifier(cell)));
                System.out.println("列值："
                + Bytes.toString (CellUtil.cloneValue(cell)));
                System.out.println("时间戳：+cell.getTimestamp());
            }
        }
        resultScanner.close(); // 关闭资源
        table.close();       // 关闭资源
    }

    public void addData(HBaseBeanhbb){
        Table table = null;
        try {
            table = conn.getTable(tbName);
        } catch (IOException e) {
            e.printStackTrace();
        }
        //实例化 Put 请求对象
        Put put = new Put(Bytes.toBytes(hbb.rowKey));
        put.addColumn(Bytes.toBytes(hbb.colFamily),
                      Bytes.toBytes(hbb.columName),
                      Bytes.toBytes(hbb.value.toString()));
        try {
            table.put(put);
            talbe.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void test(){
        this.getData("log2");
        try {
            this.testScan();
        } catch (IOException e) {
            e.printStackTrace();
        }
        this.createTable();
    }

    public static void main(String[] args) {
        HBaseDaohbd = new HBaseDao();
        hbd.test();
    }
}
```

上例代码只是对 HBase 的基本操作，更多操作需要读者自行学习，这里只是入门实例。值得读者注意的是，HBase1.0 相对 HBase0.98 API 有不少变化，使用时需要无比注意，否则会引发一些不必要的错误。

6.5 总结

本章主要向读者介绍当前键值对文档型NoSQL代表Mongo3.0和基于列簇方式NoSQL代表HBase 1.0的新特性，并对二者做了简单对比。

Mongo3.0使用WT存储引擎替代默认方式后，单表数据上亿时，在创建好相关索引时，查询性能仍然非常出色，这点是远非Mongo2.0所能比较的。Mongo3.0是Mongo发展史上的一个质的飞跃；HBase1.0同样是HBase团队七年磨一剑的产物，各方面性能也有极大提升，所以，本章向读者介绍Mongo3.0和HBase1.0，重点向读者介绍Mongo集群特性及常用索引创建方式，介绍HBase底层实现原理和系统架构，最后结合完整实例，分别给读者介绍Mongo3.0和HBase1.0的使用，尤其针对Mongo3.0使用，代码实例中展示了同步调用和异步调用返回结果的不同，希望通过阅读本章让读者对Mongo3.0和HBase1.0有一个基本认识，更能熟练使用。

第 7 章 全文检索：ElasticSearch2.x

谈到搜索（即全文检索），目前基于 Java 语言最为著名的开源项目无非是基于 Lucene 基础之上的 Solr/SolrCloud 和 Elasticsearch（简称 ES），二者可以说是目前 Apache 旗下炙手可热的顶级开源项目，目前发展都非常迅速，本章向读者介绍二者，重点向大家介绍 ES。

7.1 认识 ElasticSearch 和 Solr

7.1.1 ElasticSearch 和 Solr 基本介绍

1. ElasticSearch

ElasticSearch 简称 ES，是一个构建在全文搜索引擎 Apache Lucene™ 基础上的实时的分布式搜索和分析引擎。它提供了一个分布式多用户能力的全文搜索引擎，基于 RESTful Web 接口。ES 是用 Java 开发的，并作为 Apache 许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于云计算中，能够达到实时搜索，稳定、可靠、快速，安装使用方便。它可以帮助你用前所未有的速度去处理大规模数据。

ES 使用 Lucene 作为内部引擎，但是在使用它做全文搜索时，只需要使用统一开发好的 API 即可，而不需要了解其背后复杂的 Lucene 的运行原理。当然 ES 并不仅仅是 Lucene 这么简单，它不但包括了全文搜索功能，还可以进行以下工作:

（1）分布式实时文件存储，并将每一个字段都编入索引，使其可以被搜索。

（2）实时分析的分布式搜索引擎。

（3）可以扩展到上百台服务器，处理 PB 级别的结构化或非结构化数据。

这么多的功能被集成到一台服务器上，你可以轻松地通过客户端或者任何你喜欢的程序语言与 ES 的 RESTful API 进行交流。

ES 的上手是非常简单的。它附带了很多非常合理的默认值，这让初学者很好地避免一上手就要面对复杂的理论，它安装好了就可以使用了，用很小的学习成本就可以变得很有生产力。随着越学越深入，还可以利用 ES 更多高级的功能，整个引擎可以很灵活地进行配置。可以根据自身需求来定制属于自己的 ES。

2．Solr

Solr 是 Apache Lucene 项目的开源企业搜索平台。其主要功能包括全文检索、命中标识、分面搜索、动态聚类、数据库集成，以及富文本（如 Word、PDF）的处理。Solr 是高度可扩展的，并提供了分布式搜索和索引复制。Solr 是最流行的企业级搜索引擎，Solr4 还增加了 NoSQL 支持。Solr 是用 Java 编写、运行在 Servlet 容器（如 Apache Tomcat 或 Jetty）的一个独立的全文搜索服务器。Solr 采用了 Lucene Java 搜索库为核心的全文索引和搜索，并具有类似 REST 的 HTTP/XML 和 JSON 的 API。Solr 强大的外部配置功能使得无需进行 Java 编码，便可对其进行调整以适应多种类型的应用程序。Solr 有一个插件架构，以支持更多的高级定制。

因为 2010 年 Apache Lucene 和 Apache Solr 项目合并，两个项目是由同一个 Apache 软件基金会开发团队制作实现的。提到技术或产品时，Lucene/Solr 或 Solr/Lucene 是一样的。

目前 ES 最新版是 2.3.4，所采用的 Lucene 版本是 5.5.0，Solr 最新版是 6.1.0，可以说自 2012 年开始，二者发展都异常迅速，往往上一个版本还没有熟悉，下个新版本就推出了，当然这本身也说明二者在当前业界使用更加广泛。

7.1.2 ES 基本概念

ES 天生支持分布式集群，支持实时索引更新，这里介绍 ES 索引集群相关概念。

- Cluster 和 Node

ES 可以以单点或者集群方式运行，以一个整体对外提供 search 服务的所有节点组成 cluster，组成这个 cluster 的各个节点叫做 node。

- Index 和 Type

二者是 ES 存储数据的地方，其中 Index 类似于关系数据库的 database，Type 类似于关系数据库中的 table。Solr 中没有 Type，只有 Index。

- Shards

索引分片，这是 ES 提供分布式搜索的基础，主要是将一个完整的 index 分成若干部分存储在相同或不同的节点上，这些组成 index 的部分就叫做 shards。

- Replicas

索引副本，ES 可以设置多个索引的副本，副本的作用：一是提高系统的容错性，当某个节点某个分片损坏或丢失时可以从副本中恢复；二是提高 ES 的查询效率，ES 会自动对搜索请求进行负载均衡。

- Recovery

数据恢复或叫数据重新分布，ES 在有节点加入或退出时会根据机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。

- Gateway

ES 索引快照的存储方式，ES 默认是先把索引存放到内存中，当内存满了时再持久化到本地硬盘。Gateway 对索引快照进行存储，当这个 ES 集群关闭再重新启动时就会从 Gateway 中读取索引备份数据。

- Discovery.zen

代表 ES 的自动发现节点机制，ES 是一个基于 P2P 的系统，它先通过广播寻找存在的节点，再通过多播协议来进行节点之间的通信，同时也支持点对点的交互。如果存在一些特殊情况，当下富余服务器都是跨网段的，这时就需要在 ES 配置文件中显示配置此项。

- Transport

代表 ES 内部节点或集群与客户端的交互方式，默认内部是使用 TCP 协议进行交互，同时它支持 HTTP 协议（JSON 格式）。

- Mapping

Mapping 非常类似于静态语言中的数据类型，与开发语言中的数据类型相比，mapping 还有一些其他的含义，mapping 不仅告诉 ES 一个 field 中是什么类型的值，它还告诉 ES 如何索引数据，以及数据是否能被搜索到。一个 mapping 由一个或多个 analyzer 组成，一个 analyzer 又由一个或多个 filter 组成的。当 ES 索引文档的时候，它把字段中的内容传递给相应的 analyzer，analyzer 再传递给各自的 filters。filter 的功能很容易理解：一个 filter 就是一个转换数据的方法，输入一个字符串，这个方法返回另一个字符串，比如一个将字符串转为小写的方法就是一个 filter 很好的例子。一个 analyzer 由一组顺序排列的 filter 组成，执行分析的过程就是按顺序一个 filter 一个 filter 地依次调用，ES 存储和索引最后得到的结果。

总起来说，mapping 的作用就是执行一系列的指令将输入的数据转成可搜索的索引项。虽然 ES 无模式，那是因为 ES 会对没有指定模式的索引数据创建默认的 Mapping。学习和测试阶段或许可以放任使用 ES 默认 Mapping，但生产环境下，

笔者建议您最好还是老老实实地把核心字段数据定义好相应 Mapping，否则会大大影响索引的搜索性能和效果。

7.1.3 ES 和 SolrCloud 集群结构

图 7.1 和图 7.2 分别为 ES2.0 和 SolrCloud5.0 和集群结构图。

图 7.1 展示了包含 2 个索引：test 和 test2 的 ES 集群，每个索引均包含 3 个 shard，且每个 shard 包含一主一备，从图中可以看出，每个索引节点中 shard 都是间隔开来，形成一个循环，这样使得集群结构更健壮。

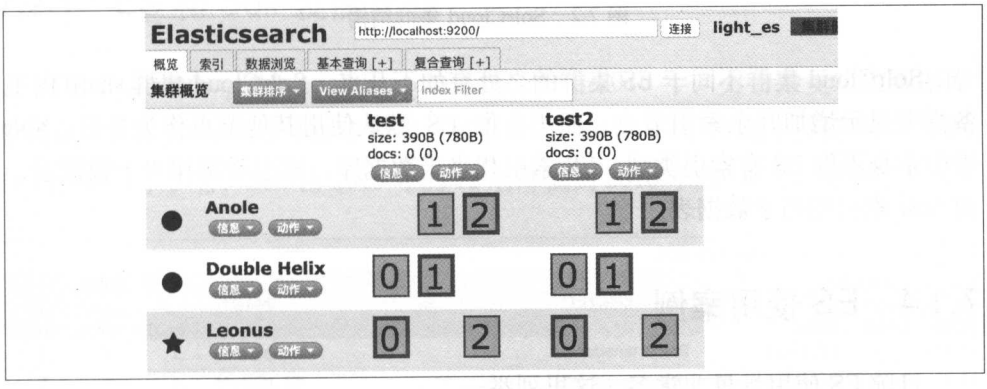


图 7.1 ES 集群结构

图 7.1 所示为 ES 图形界面管理工具 Head 插件，通过该插件可以方便的进行索引管理，比如索引的创建（Shard 及 relpica 个数控制）、删除、查询分析等。此外还有一个 ES 插件 bigdesk，利用它可以方便 ES 管理员监控每个 ES 节点日常工作状态：负载、网络流量等实时情况。ES 集群搭建相对比较简单，不需要依赖额外第三方，因此更加易用、健壮。

图 7.2 展示了包含 3 个索引：house、news 和 jiaju 的 SolrCloud 集群，且每个索引均包含 3 个 shard，完整的集群应该像上述 ES 集群那样，每个 Shard 至少包含一主一备两个节点，这样集群整体健壮性更好。这里限于实验机器个数，每个 Shard 内仅有单个节点，没有做主备。SolrCloud 本身是一个 Web 应用，其集群的搭建需要依赖 Tomcat 和 Zookeeper 第三方组件，每一个 SolrCloud 节点都需要依

赖一个 Tomcat 节点。

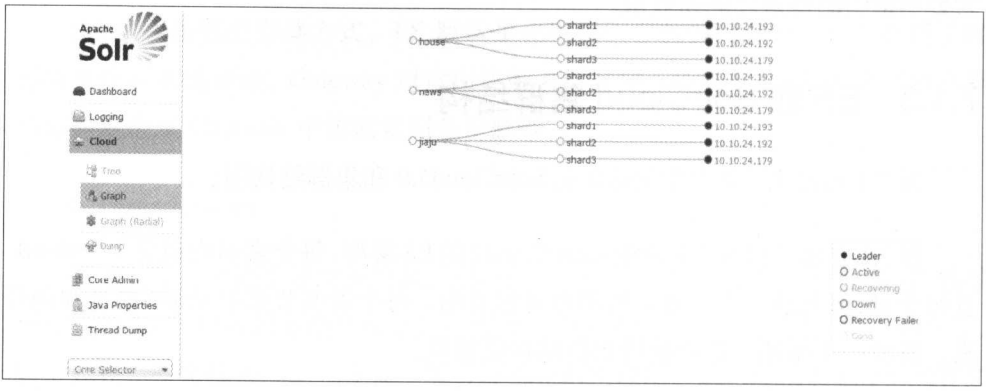


图 7.2 SolrCloud 集群结构

SolrCloud 集群不同于 ES 集群的之处有如下几点：SolrCloud 集群 shard 内主备需要显示增加冗余索引节点，而不会像 ES 默认使用其他节点作为备份；Solr 索引本身不像 ES 有索引类型，ES 索引相当于数据库，索引类型相当于数据表，而 Solr 索引相当于数据表。

7.1.4 ES 使用案例

目前 ES 使用场景非常多，这里列举一二。

（1）维基百科使用 Elasticsearch 来进行全文搜做并高亮显示关键词，以及提供 search-as-you-type、did-you-mean 等搜索建议功能。

（2）英国卫报使用 Elasticsearch 来处理访客日志，以便能将公众对不同文章的反应实时地反馈给各位编辑。

（3）StackOverflow 将全文搜索与地理位置和相关信息进行结合，以提供 more-like-this 相关问题的展现。

（4）GitHub 使用 Elasticsearch 来检索超过 1300 亿行代码。

（5）每天，Goldman Sachs 使用它来处理 5TB 数据的索引，还有很多投行使用它来分析股票市场的变动。

Elasticsearch 并不只是面向大型企业的，它还帮助了很多类似 DataDog、Klout，以及国内很多的创业公司进行了功能的扩展。

7.2 ES 和 Solr 比较分析

ES 和 Solr 均是当前最为流行的全文检索工具，且二者都基于 Lucene 延伸而来，但二者还是有很大不同的，综合对比，ES 比 Solr 更适合做企业级搜索平台，作者本人也是从 Solr 转向 ES 开发，下面对比介绍二者。

7.2.1 ES 和 Solr 发展比较

DB-Engines(http://db-engines.com)最近公布了 2016 年 7 月份最受欢迎数据库管理系统，图 7.3 给出了 Top15 排名，其中 ES 由 2015 年的 14 名上升到 2016 年的 11 名，提升了 3 名；Solr 由 2015 年的 12 名降低到 2016 年的 14 名，降低了 2 名。

308 systems in ranking, July 2016

Rank			DBMS	Database Model	Score		
Jul 2016	Jun 2016	Jul 2015			Jul 2016	Jun 2016	Jul 2015
1.	1.	1.	Oracle	Relational DBMS	1441.53	-7.72	-15.20
2.	2.	2.	MySQL	Relational DBMS	1363.29	-6.85	+79.95
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1192.89	+27.08	+89.83
4.	4.	4.	MongoDB	Document store	315.00	+0.38	+27.61
5.	5.	5.	PostgreSQL	Relational DBMS	311.15	+4.55	+38.33
6.	6.	6.	DB2	Relational DBMS	185.08	-3.49	-13.04
7.	7.	↑ 8.	Cassandra	Wide column store	130.70	-0.42	+17.99
8.	8.	↓ 7.	Microsoft Access	Relational DBMS	124.90	-1.32	-19.40
9.	9.	9.	SQLite	Relational DBMS	108.53	+1.75	+2.66
10.	10.	10.	Redis	Key-value store	108.03	+3.54	+12.96
11.	11.	↑ 14.	Elasticsearch	Search engine	88.62	+1.21	+18.46
12.	12.	↑ 13.	Teradata	Relational DBMS	73.93	+0.10	+1.62
13.	13.	↓ 11.	SAP Adaptive Server	Relational DBMS	70.73	-0.95	-16.48
14.	14.	↓ 12.	Solr	Search engine	64.69	+0.63	-14.59
15.	15.	15.	HBase	Wide column store	53.14	+0.15	-7.77

图 7.3 2016 年 7 月 Top15 DB-Engins 排名

图 7.3 的排名是把 Elasticsearch 和全球的数据库公司进行了对比，看起来还是有点吃亏的，我们换个角度，只把 Elasticsearch 看成全文搜索库，我们来和 Solr

对比一下就可以发现更有意思的事情，如图 7.4 所示。

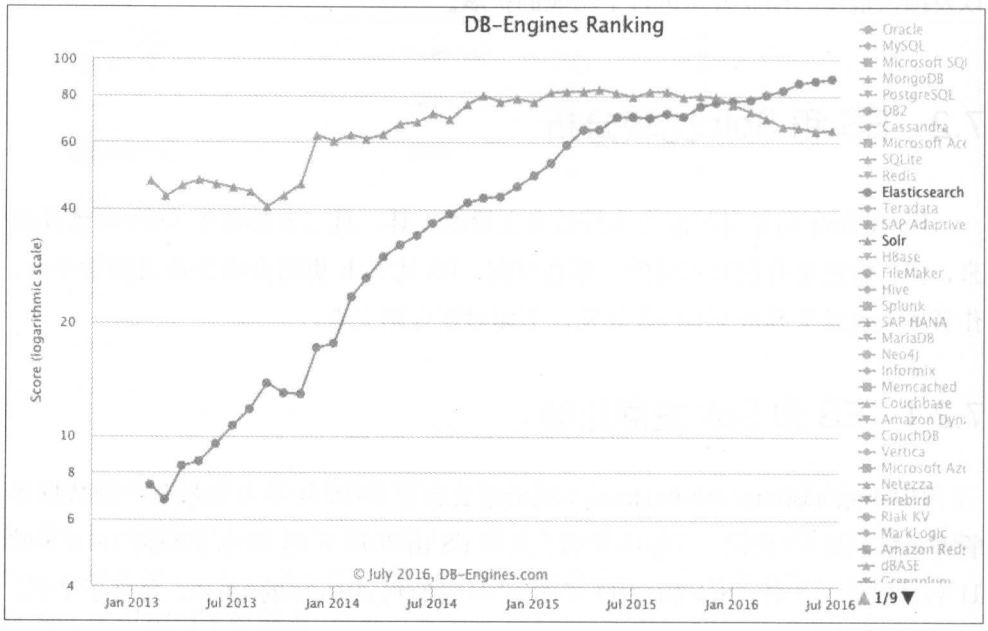


图 7.4 ES 和 Solr 近三年受欢迎发展曲线

你可以发现，我们可以发现在 2013 年，在全文搜索领域，Solr 显然占主导地位，但随着时间的推移，Elasticsearch 在迎头赶上，直到 2016 年年初，Elasticsearch 已经超越了 Solr，小幅领先，而且看发展趋势，ES 领先 Solr 的幅度有越来越大之势。从这里也可以看出 ES 更加符合当前市场需要。

7.2.2 ES 和 Solr 综合比较

1. 索引性能

为了充分测试 ES 和 Solr 在索引读写及负载均衡方面的效果对比分析，作者分别搭建了一套 ES 和 SolrCloud 集群，集群结构如图 7.1 和图 7.2 所示，两集群均包含 3 个 Shard。在没有做任何优化配置前提下，对二集群均采用单进程和单线程方式插入索引或查询数据如下，分别向 2 个集群写入 10 万条数据（每条数据包含 4 个字段），这里向读者介绍作者本人的实验结果，如表 7.1 所示的 ES 和 Solr

索引性能对比。

表 7.1 ES 和 Solr 索引性能对比

比 较 项	ES	Solr
写索引耗时	新插入 10 万条耗时 18.8 秒 更新 2 万插入 8 万耗时 64 秒	新插入 10 万条耗时 7.2 秒 更新 2 万插入 8 万耗时 24 秒
读索引耗时	**返 10 万条数据大概 15 毫秒**	**返 10 万条数据大概 29 毫秒**
Shard 数据个数差异	差异相对较大，个数可达千	差异相对较小，个数仅有几十
索引文件大小	**索引文件相对较大（7612KB）**	**索引文件相对较小（5280KB）**

此外，当单纯地对已有数据进行搜索时，Solr 更快；当实时建立索引时，Solr 会产生 IO 阻塞，查询性能较差，Elasticsearch 具有明显的优势。随着数据量的增加，Solr 的搜索效率会变得更低，而 Elasticsearch 却没有明显的变化。

2. 健壮性

图 7.1 和图 7.2 分别展示了 SolrCloud 和 ES 的集群结构，集群搭建过程中，SolrCloud 本身需要以来第三方 Web 容器（Jetty 或 Tomcat）、需要借助 Zookeeper 进行分布式管理；而 ES 本身采用 Netty 作为默认的 Http 容器，而且自身就具有分布式协调管理能力。

生产环境下，集群基础环境所依赖的软件越多，其出错的可能性就越大，而且集群搭建更显得复杂，所以相比较，ES 在健壮性层面比 Solr 更优。

3. 容错性

对于含有 3 个 Shard、6 个 Node 的集群结构而言，如果 Node 故障数不超过 3 个，那么 SolrCloud 和 ES 集群都可以正常提供服务，而当 Node 节点故障数超过分片 Shard 个数 3 时，假如有 4 个 Node 节点故障，这是 SolrCloud 集群已经不能正常提供服务，而 ES 集群还可以提供基本查询功能，只是会丢掉故障所在 Shard 分区数据。由此可以看出，容错行层面，ES 比 Solr 更优。

这点在生产环境下特别重要，设想一下：假如真的出现上述问题，作为服务方，相信大家肯定更希望继续让用户看到一些数据，而不是一个 500 错误页面或

空白提示搜索故障的画面。

4．中文分词支持

作为当前知名开源全文检索工具，ES 和 Solr 同宗同源，底层均构建在 Lucene 之上，目前二者对中文分词器支持都比较好，而且支持的中文分词器有很多开源项目，比如 MMSEG、IK－Analyzer、Ansj 等，还有已经内置到 Lucene 底层的 SmartCN 等。虽然目前知名中文分词器对二者都支持很好，而且也都支持专有词库扩展，但相比较而言，生产环境下，用户更希望能够实时扩展分词专有词库，而不至于对线上索引集群造成过大影响（比如，需要重启索引节点才可以完成专有词库扩展等）。遗憾的是，目前 Solr 还不支持动态调整专有词库（如果想实现此功能，需要用户自己定制开发），而 ES 则有很好的支持，比如 ES 中文分词器插件 IK-Analyzer 和 Ansj 等，都可以实时动态调整专有词库，而不需要重启 ES 相关 Node 节点，二者实现方式分别如下：

IK-Analyzer 通过 ES 各个 Node 节点每隔 10 秒钟监听远程专有词库文件是否修改，根据最近更新时间是否变化（各个 Node 节点会保存上一次最近修改时间），来决定在 Node 节点是否重新加在远程专有扩展词库，从而完成动态实时更新专有词。

Ansj 则通过使用 Redis 的 Pub-Sub 的 MQ 功能，各个 Node 节点，通过实时监听 Redis 中事先设置好的广播模式，来完成各个 Node 节点中的专有词库动态调整。

综上所述，虽然 ES 在索引有些方面不如 Solr，但整体看，ES 要优于 Solr；而且从 DB-Engins 排名看，也很明显，近 3 年来，ES 使用广度增长速度远超 Solr。值得说明一点，笔者本人在 2014 年之前一直使用 Solr/SolrCloud，对 ES 不屑，后来因工作变动，周遭同事都用 ES，因此也入乡随俗，试着用了一把，结果一发不可收拾，至现在，笔者所负责的相关搜索项目一直都是首选 ES，笔者本人就是一个从 Solr 转向 ES 的典型开发者。

因此，作者个人建议生产环境下使用 ES 替代 Solr。

7.3 ES 集群介绍

最近两年，ES 发展异常迅速，为方便大家学习掌握，这里以当前最新版 ES2.2.1 为例向大家介绍 ES 集群搭建相关过程。

7.3.1 插件安装

ES 本身关注核心功能点的开发，相关辅助功能均通过插件方式提供，比如 ES 管理所依赖的 Head、BigDesk，以及中文分词等，这里重点向大家介绍方便对 ES 进行各种操作的客户端 Head 插件。

Head 插件安装主要参考其官方 Git 网址：https://github.com/mobz/elasticsearch-head，里面有详细说明，这里向大家介绍其注意事项，因为随着 ES 快速迭代，Head 插件本身也在快速更新，因此不同大的版本 ES 其对应的 Head 插件安装方式也有不同：

- 对于 ES2.x 版本

执行命令：sudo {ES_HOME}/bin/plugin install mobz/elasticsearch-head

- 对于 ES1.x 版本

执行命令：sudo {ES_HOME}/bin/plugin install mobz/elasticsearch-head/1.x

- 对于 ES0.9 版本

执行命令：sudo {ES_HOME}/bin/plugin install mobz/elasticsearch-head/0.9

下面是作者在自己笔记本为 ES2.2.1 安装最新版 Head 的执行过程：

```
    localhost:es221 a$ bin/plugin install mobz/elasticsearch-head
    -> Installing mobz/elasticsearch-head...
    Trying
https://github.com/mobz/elasticsearch-head/archive/master.zip ...
    Downloading ..................................................
..............................................................
..............................................................
..............................................................
```

```
.....
.....
.....
.....
.....
.....DONE
    Verifying
https://github.com/mobz/elasticsearch-head/archive/master.zip
checksums if available ...
    NOTE: Unable to verify checksum for downloaded plugin (unable to
find .sha1 or .md5 file to verify)
    Installed head into /Users/a/Desktop/jobs/es221/plugins/head
```

至此，Head 插件安装完毕，${ES_HOME}目录下会多了一个 plugins 目录，其下有刚安装好的 head 目录，即对应 head 插件。

7.3.2 中文分词安装

支持 ES 的中文分词插件有很多，比如 MMSEG、Ansj 和 IK-Analyzer 等，这里作者以过去项目实践经验，建议大家使用 IK-Analyzer，其 ES 相应插件 es-analysis-ik 网址：https://github.com/medcl/elasticsearch-analysis-ik/tree/master。

目前支持 ES2.2.1 版的 elasticsearch-analysis-ik-1.8.1，下载该版本源码包，解压后，进入分词插件工程目录下，编译如下：

```
    localhost:elasticsearch-analysis-ik-1.8.1 a$ mvn clean install
    [INFO] Scanning for projects...
    [INFO]
    [INFO] ------------------------------------------------------------
    [INFO] Building elasticsearch-analysis-ik 1.8.0
    [INFO] ------------------------------------------------------------
    ......
    [INFO] Installing /Users/a/Desktop/jobs/elasticsearch-analysis-
ik-1.8.1/target/releases/ elasticsearch-analysis-ik-1.8.0.zip to
/Users/a/.m2/repository/org/elasticsearch/ elasticsearch-
analysis-ik/1.8.0/elasticsearch-analysis-ik-1.8.0.zip
    [INFO] ------------------------------------------------------------
    [INFO] BUILD SUCCESS
    [INFO] ------------------------------------------------------------
    [INFO] Total time: 15:56 min
    [INFO] Finished at: 2016-03-28T14:31:26+08:00
    [INFO] Final Memory: 33M/503M
    [INFO] ------------------------------------------------------------
```

编译成功，现在把 elasticsearch-analysis-ik-1.8.1 编译好的文件移到 es 相关插

件下即可，即：

```
    mv      ${ES_IK_HOME}/target/releases/elasticsearch-analysis-ik-
1.8.0/* ${ES_HOME}/plugins/ik/。
```

IK-Analyzer 有两种分词模式：智能切分 ik_smart 和细粒度迭代最大切分 ik_max_word（即默认方式：ik），现在验证一下中文分词效果：

- 使用 ES 自带的标准分词 standard 方式

```
    http://localhost:9200/_analyze?analyzer=standard&text=中华人民共和
国，结果如下：
    {"tokens":[{"token":"中","start_offset":0,"end_offset":1,"type":
"<IDEOGRAPHIC>","position":0},
    {"token":"      华      ","start_offset":1,"end_offset":2,"type":
"<IDEOGRAPHIC>","position":1},
    {"token":"      人      ","start_offset":2,"end_offset":3,"type":
"<IDEOGRAPHIC>","position":2},
    {"token":"      民      ","start_offset":3,"end_offset":4,"type":
"<IDEOGRAPHIC>","position":3},
    {"token":"      共      ","start_offset":4,"end_offset":5,"type":
"<IDEOGRAPHIC>","position":4},
    {"token":"      和      ","start_offset":5,"end_offset":6,"type":
"<IDEOGRAPHIC>","position":5},
    {"token":"      国      ","start_offset":6,"end_offset":7,"type":
"<IDEOGRAPHIC>","position":6}]}
```

- 使用 IK-Analyzer 的 ik_smart 方式

```
    http://localhost:9200/_analyze?analyzer=ik_smart&text=中华人民共和
国
    {"tokens":[
    {"token":"中华人民共和国","start_offset":0,
    "end_offset":7,"type":"CN_WORD","position":0}]}
```

- 使用 IK-Analyzer 的 ik 方式

```
    http://localhost:9200/_analyze?analyzer=ik&text=中华人民共和国
    {"tokens":[
    {"token":"中华人民共和国","start_offset":0,"end_offset":7,"type":
"CN_WORD","position":0},
    {"token":"中华人民","start_offset":0,"end_offset":4,"type":
"CN_WORD","position":1},
    {"token":"中华","start_offset":0,"end_offset":2,"type":
"CN_WORD","position":2},
    {"token":"华人","start_offset":1,"end_offset":3,"type":
```

```
"CN_WORD","position":3},
    {"token":"人民共和国","start_offset":2,"end_offset":7,"type":
"CN_WORD","position":4},
    {"token":"人民","start_offset":2,"end_offset":4,"type":
"CN_WORD","position":5},
    {"token":"共和国","start_offset":4,"end_offset":7,"type":
"CN_WORD","position":6},
    {"token":"共和","start_offset":4,"end_offset":6,"type":
"CN_WORD","position":7},
    {"token":"国","start_offset":6,"end_offset":7,"type":
"CN_CHAR","position":8}]}
```

7.3.3 ES2.X 集群节点类型

在 Elasticsearch 中节点可以分为主（master）节点、数据（data）节点、客户端节点和部落节点[1]，每种类型的节点有不同的使用方法，对于一个大的集群中，合理地配置这些属性，对集群的健壮性和性能有很大的帮助。

当我们启动 Elasticsearch 的实例，就会启动至少一个节点。相同集群名的多个节点的连接就组成了一个集群，在默认情况下，集群中的每个节点都可以处理 http 请求和集群节点的数据传输，集群中所有的节点都知道集群中其他所有的节点，可以将客户端请求转发到适当的节点。节点有以下类型：

1. 主（master）节点

在一个节点上当 node.master 设置为 True（默认）的时候，它有资格被选作为主节点，控制整个集群。主节点的主要职责是和集群操作相关的内容，如创建或删除索引，跟踪哪些节点是群集的一部分，并决定哪些分片分配给相关的节点。稳定的主节点对集群的健康是非常重要的。默认情况下任何一个集群中的节点都有可能被选为主节点。索引数据和搜索查询等操作会占用大量的 CPU、内存、IO 资源，为了确保一个集群的稳定，分离主节点和数据节点是一个比较好的选择。

虽然主节点也可以协调节点，路由搜索和从客户端新增数据到数据节点，但

1 Elasticsearch2.2.0 节点类型详解 http://my.oschina.net/secisland/blog/618911?fromerr=cJzlBvKR

最好不要使用这些专用的主节点。一个重要的原则是，尽可能做尽量少的工作。创建一个独立的主节点的配置为：

```
node.master:true
node.data: false
```

为了防止数据丢失，配置 discovery.zen.minimum_master_nodes 设置是至关重要的（默认为 1），每个主节点应该知道形成一个集群的最小数量的主资格节点的数量。

假设一个集群，有 3 个主节点，当网络发生故障的时候，有可能其中一个节点不能和其他节点进行通信了。这个时候，当 discovery.zen.minimum_master_nodes 设置为 1 的时候，就会分成两个小的独立集群，当网络好的时候，就会出现数据错误或者丢失数据的情况。当 discovery.zen.minimum_master_nodes 设置为 2 的时候，一个网络中有两个主资格节点，可以继续工作，另一部分，由于只有一个主资格节点，则不会形成一个独立的集群，这个时候当网络回复的时候，节点又会重新加入集群。设置这个值的原则是：

```
(master_eligible_nodes / 2) + 1。这个参数也可以动态设置：
PUT _cluster/settings{
  "transient": {
    "discovery.zen.minimum_master_nodes": 2
  }
}
```

2. 数据（data）节点

在一个节点上 node.data 设置为 True（默认）的时候。该节点保存数据和执行数据相关的操作，如增删改查、搜索和聚合。数据节点主要是存储索引数据的节点，主要对文档进行增删改查操作、聚合操作等。数据节点对 CPU、内存、IO 要求较高，在优化的时候需要监控数据节点的状态，当资源不够的时候，需要在集群中添加新的节点。

数据节点的配置如下：

```
node.master: false
node.data: true
```

数据目录可以被多个节点共享，甚至可以属于不同的集群，为了防止多个节点共享相同的数据路径，可以在配置文件 elasticsearch.yml 中添加：

```
node.max_local_storage_nodes: 1
```

注意：在相同的数据目录不要运行不同类型的节点(例如：master, data, client)，这会导致意外的数据丢失。

3．客户端节点

当一个节点的 node.master 和 node.data 都设置为 false 的时候，它既不能保持数据也不能成为主节点，该节点可以作为客户端节点响应用户的情况，并把相关操作发送到其他节点。当主节点和数据节点配置都设置为 false 的时候，该节点只能处理路由请求、处理搜索、分发索引操作等，从本质上来说，该客户节点表现为智能负载平衡器。独立的客户端节点在一个比较大的集群中是非常有用的，它协调主节点和数据节点，客户端节点加入集群可以得到集群的状态，根据集群的状态可以直接路由请求。

警告：添加太多的客户端节点对集群是一种负担，因为主节点必须等待每一个节点集群状态的更新确认！客户节点的作用不应被夸大，数据节点也可以起到类似的作用。配置如下：

```
node.master: false
node.data: false
```

4．部落节点

当一个节点配置 tribe.*的时候，它是一个特殊的客户端，可以连接多个集群，在所有连接的集群上执行搜索和其他操作。部落节点可以跨越多个集群，它可以接收每个集群的状态，然后合并成一个全局集群的状态，它可以读写所有节点上的数据，部落节点在 elasticsearch.yml 中的配置如下：

```
tribe:
    t1:
        cluster.name:   cluster_one
    t2:
        cluster.name:   cluster_two
```

t1 和 t2 是任意的名字代表连接到每个集群。上面的示例配置两集群连接，名称分别是 t1 和 t2。默认情况下部落节点通过广播可以作为客户端连接每一个集群。大多数情况下，部落节点可以像单节点一样对集群进行操作。

注意：

以下操作将和单节点操作不同：

- 如果两个集群的名称相同，部落节点只会连接其中一个。
- 由于没有主节点，当设置 local 为 true 的是，主节点的读操作会被自动执行，例如：集群统计、集群健康度。
- 主节点级别的写操作将被拒绝，这些应该在一个集群进行。

部落节点可以通过块（block）设置所有的写操作和所有的元数据操作，例如：

```
tribe:
   blocks:
      write:    true
      metadata: true
```

部落节点也可以在选中的索引块中进行配置，例如：

```
tribe:
   blocks:
      write.indices:    hk*,ldn*
      metadata.indices: hk*,ldn*
```

当多个集群有相同的索引名的时候，默认情况下，部落的节点将选择其中一个。这可以通过 tribe.on_conflict setting 进行配置，可以设置排除那些索引或者指定固定的部落名称。

默认情况下，节点配置是一个主节点和一个数据节点。这是非常方便的小集群，但随着集群的发展，分离主节点和数据节点将变得非常重要。

节点协调、搜索请求或批量增加索引请求等可能涉及在不同的数据节点上操作。在这些请求会分成两个阶段：一是接收客户端的请求，二是协调节点执行相关操作。当数据分散在不同的节点上的时候，协调节点将请求转发到数据节点，每个数据节点在本地执行请求并把结果传输给协调节点，然后协调节点收集各个

数据节点的结果转换成一个单一的请求结果返回。所以需要客户端有足够的内存和 CPU 来处理各个节点的返回结果。

7.3.4 ES 配置事项

ES 集群搭建及今后运维过程中，有很多事项需要大家小心，否则会影响 ES 平台集群稳定性和查询及索引性能。

1. 内存优化配置

ES 默认安装后设置的内存是 1GB，对于任何一个业务部署来说，这个都太小了，这样的集群配置可能有点问题[2]。这里有两种方式修改 Elasticsearch 的堆内存（下面就说内存好了），最简单的一个方法就是指定 ES_HEAP_SIZE 环境变量。服务进程在启动时候会读取这个变量，并相应的设置堆的大小。举例，你可以用下面的命令设置它：

```
export ES_HEAP_SIZE=10g
```

此外，你也可以通过命令行参数的形式，在程序启动的时候把内存大小传递给它：

```
./bin/elasticsearch -Xmx10g -Xms10g
```

注意：确保 Xmx 和 Xms 的大小是相同的，防止程序在运行时改变大小，否则是很废资源的。Heap 分配多少合适？遵从官方建议就没错。不要超过系统可用内存的一半，并且不要超过 32GB。这里分别解释一下。

内存对于 Elasticsearch 来说绝对是重要的，用于更多的内存数据提供更快的操作。而且还有一个内存消耗大户——Lucene。Lucene 的设计目的是把底层 OS 里的数据缓存到内存中。Lucene 的段是分别存储到单个文件中的，这些文件都是不会变化的，所以很利于缓存，同时操作系统也会把这些段文件缓存起来，以便更快地访问。Lucene 的性能取决于和 OS 的交互，如果你把所有的内存都分配给 Elasticsearch，不留一点给 Lucene，那你的全文检索性能会很差。最后标准的建议

2 ElasticSearch 内存分配设置详解 http://openskill.cn/article/304

是把 50%的内存给 Elasticsearch，剩下的 50%也不会没有用处，Lucene 会很快吞噬剩下的这部分内存。

另外一个原因不分配大内存给 Elasticsearch，事实上 jvm 在内存小于 32GB 的时候会采用一个内存对象指针压缩技术。在 Java 中，所有的对象都分配在堆上，然后有一个指针引用它。指向这些对象的指针大小通常是 CPU 的字长的大小，不是 32bit 就是 64bit，这取决于你的处理器，指针指向了值的精确位置。对于 32 位系统，你的内存最大可使用 4GB。对于 64 系统可以使用更大的内存。但是 64 位的指针意味着更大的浪费，因为你的指针本身大了。浪费内存不算，更糟糕的是，更大的指针在主内存和缓存器（例如 LLC, L1 等）之间移动数据的时候，会占用更多的带宽。

Java 使用一个叫内存指针压缩的技术来解决这个问题。它的指针不再表示对象在内存中的精确位置，而是表示偏移量。这意味着 32 位的指针可以引用 40 亿个对象，而不是 40 亿个字节。最终，也就是说，堆内存长到 32GB 的物理内存，也可以用 32bit 的指针表示。一旦你越过那个神奇的 30GB~32GB 的边界，指针就会切回普通对象的指针，每个对象的指针都变长了，就会使用更多的 CPU 内存带宽，也就是说你实际上失去了更多的内存。事实上当内存到达 40GB~50GB 的时候，有效内存才相当于使用内存对象指针压缩技术时候的 32GB 内存。这段描述的意思就是说：即便你有足够的内存，也尽量不要超过 32GB，因为它浪费了内存，降低了 CPU 的性能，还要让 GC 应对大内存。

2．超大搜索聚合结果集的Fetch

ES 是分布式搜索引擎，搜索和聚合计算除了在各个 data node 并行计算以外，还需要将结果返回给汇总节点进行汇总和排序后再返回。无论是搜索，还是聚合，如果返回结果的 size 设置过大，都会给 heap 造成很大的压力，特别是数据汇聚节点。超大的 size 多数情况下都是用户用例不对。如果确实需要大量拉取数据可以采用 scan & scroll api 来实现。

3．熟悉各类缓存作用

要对如 field cache, filter cache, indexing cache, bulk queue 等等，设置合理的大

小，并且要应该根据最坏的情况来看 heap 是否够用。

4．合理设置shard和replica

根据机器数、磁盘数、索引大小等硬件环境，根据测试结果，设置最优的分片数和备份数，单个分片最好不超过 10GB，定期删除不用的索引，做好冷数据的迁移。

5．倒排词典的索引需要常驻内存

无法 GC，需要监控 data node 上 segment memory 增长趋势。

6．保守配置内存限制参数

尽量使用 doc value 存储以减少内存消耗，查询时限制 size、from 参数。

7.4 ES 客户端使用

ES 除了借助 Head 等插件直接创建、删除和查询相关索引外，还可以通过 ES 的各个开发语言客户端管理 ES，这里以 Java 语言为例向大家介绍。其中 ES 开发的核心 Maven 依赖包如下：

```
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>2.3.3</version>
</dependency>
```

此外，还需要增加 Groovy 是当前 ES 默认支持的脚本引擎，其依赖如下：

```
<dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-all</artifactId>
    <version>2.3.2</version>
    <classifier>indy</classifier>
</dependency>
```

如果开发中涉及地理空间索引，需要增加一下依赖：

```
<!——Spatial4j 是一个通用的空间/地理空间 ASL 许可的开源 Java 库—->
<dependency>
    <groupId>com.spatial4j</groupId>
    <artifactId>spatial4j</artifactId>
    <version>0.4.1</version>
</dependency>
<dependency>
    <groupId>com.vividsolutions</groupId>
    <artifactId>jts</artifactId>
    <version>1.13</version>
    <exclusions>
        <exclusion>
            <groupId>xerces</groupId>
            <artifactId>xercesImpl</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

7.4.1 ES 客户端连接

我们可以把 ES 看作一个特殊的数据库，数据库操作之前，离不开数据库连接，同样，ES 操作前也需要首先获得 ES 的客户端连接。这里我们提供一个 ES 客户端连接工具，完整代码参考如下：

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.Settings;
import
org.elasticsearch.common.transport.InetSocketTransportAddress;
import org.elasticsearch.common.transport.TransportAddress;
import org.light.rtc.util.Constants;

public class EsUtil {
    private static EsUtilinstace;
    //单例模式
    public static EsUtilgetInstance(){
        if(instace==null){
            instace = new EsUtil();
        }
        return instace;
    }
    //ES 集群基本设置：影响索引性能
    static Settings settings = Settings.settingsBuilder()
                //ES 集群名
                  .put("cluster.name", "light_es")
             //ES 索引刷新文档阈值：每 10000 刷新一次
            .put("index.translog.flush_ threshold_ops","10000")
```

```
                    //ES 索引刷新周期：每 2 秒刷新一次
                    .put("index.refresh_interval", "2s")
                    .put("client.transport.sniff", true).build();

        private static TransportClient client;
        static {
            try {
               client = TransportClient.builder().settings(settings).
build();
                //配置文件中配置好的集群 Host 及 port
                String[] iPports =
                              Constants.esClusterHosts. split(",");
                TransportAddress[] tas =
                     new InetSocketTransportAddress [iPports.length];
                for(int i=0; i<iPports.length; i++){
                String[] ipPort = iPports[i].split(":");
                    //声明索引节点 IP+端口
                tas[i] = new InetSocketTransportAddress(
                            InetAddress.getByName(ipPort[0]),
                            Integer. parseInt(ipPort[1]));
            }
                client.addTransportAddresses(tas);
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
        //获取
        public synchronized TransportClientgetClient() {
            return client;
        }
    }
```

进行操作前，首先工作 ESUtil 获取 ES 连接，代码如下：

```
TransportClient client = EsUtil.getInstance().getClient();
```

其他操作 ES 之处，只要使用这个 client 即可。

7.4.2 ES 基本操作

1. Mapping映射

本例以之前项目中所设计的一个文档关键词索引定义为例，向大家加以介绍。

```
import java. io. IOException;

import static org.elasticsearch.common.xcontent.XContentFactory.
jsonBuilder;
```

```
    import org. elasticsearch. action. admin. indices. mapping. put.
PutMappingRequest;
    import org. elasticsearch. client. Requests;
    import org. elasticsearch. client. transport. TransportClient;
    import org. elasticsearch. common. xcontent. XContentBuilder;

    import org. light. rtc. util. Constants;
    //文档索引
    public class NewsIndex  {
        //创建索引入口
        public void createKwIndex(){
            //通过ES客户端工具获取ES客户端连接
            TransportClientesClient = EsUtil.getInstance().getClient();
            XContentBuilder mapping = null;
            try {
                mapping = jsonBuilder().startObject()
                    .startObject ("properties")
                    //主键_id，为系统默认键，这里值为关键词ID
                    //关键词修改时间，长整型，创建索引但不分词，存储在ES中
                .startObject("updateTime").field("type", "long")
                        .field("index","not_analyzed")
                        .field ("store",true).endObject()
                //关键词名，字符串型，索引且分词，索引创建时使用分词器IK,
                //查询索引时使用分词器ik_smart，存储在ES中
                .startObject("kwdName").field("type", "string")
                        .field("analyzer","ik")
                        .field("search_analyzer","ik_smart")
                        . field("store",true).endObject()
                    //标题，字符串型，索引且分词，索引创建时使用分词器IK,
                    //查询索引时使用分词器ik_smart，存储在ES中
                .startObject("title").field("type", "string")
                        .field ("analyzer","ik")
                        .field("search_analyzer","ik_smart")
                        .field ("store",true).endObject()
                    //关键词存活周期，整型，创建索引但不分词，存储在ES中
                .startObject("timeLiness").field("type", "integer")
                        .field("index","not_analyzed")
                        .field("store", true).endObject()
                    //关键词阈值，浮点型，创建索引但不分词，存储在ES中
                    .startObject("kwdThresh").field("type", "float")
                        .field("index","not_analyzed")
                        .field("store", true).endObject()
                //嵌套结构字段：相关关键词
                    .startObject("relKwd").startObject("properties")
                        //关键词ID，字符串，不创建索引，存储在ES
                        .startObject("kwId").field("type", "string")
                            .field("index","no")
                            .field("store",true). endObject()
                        //关键词名，字符串，不创建索引，存储在ES
                    .startObject("kwName").field("type", "string")
                            .field("index","no")
```

```
                    .field("store",true). endObject()
                    //关键词权重，字符串，不创建索引，存储在ES
                .startObject("weight").field("type", "float")
                        .field("index","no")
                        .field("store", true).endObject()
                .endObject().endObject()
            .endObject().endObject();
        } catch (IOException e) {
            e.printStackTrace();
        }
        //创建索引Mapping请求
        PutMappingRequestmappingRequest = Requests
                .putMappingRequest(Constants. keyowrdsIndexName)
                .type(Constants. keywordsIndexType)
                .source(mapping);
        esClient.admin().indices()
                .putMapping (mappingRequest).actionGet();
        esClient = null;
    }

    public void run(){
        this.createKwIndex();
    }

    public static void main(String[] args) {
        KeywordsIndexkwi = new KeywordsIndex();
        kwi.run();
    }
}
```

这里需要说明的是，以ES V2.0.0.beta1为分界线，如果使用特定中文分词器，在对某个文本字段索引时，如果想在创建索引和查询时分别使用不同分词器类型，那么以IK为例：

使用之前版本ES时，形如如下格式：

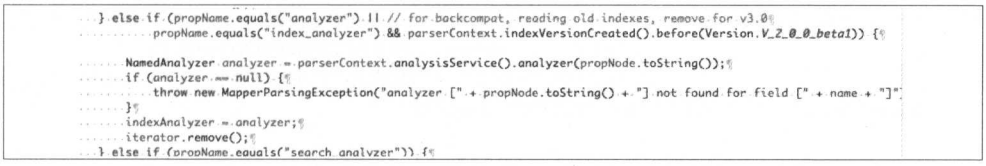
```
.startObject("title").field("type", "string")
                     .field("index analyzer","ik")
                     .field("search_analyzer","ik_smart")
                     .field("store",true).endObject()
```

使用之后版本ES时，形如如下格式：

```
.startObject("title").field("type", "string")
                     .field("analyzer","ik")
                     .field("search_analyzer","ik_smart")
                     .field("store",true).endObject()
```

具体原因可以参考其官方源代码（如图 7.5 所示）：

https://github.com/elastic/elasticsearch/blob/master/core/src/main/java/org/elasticsearch/index/mapper/core/TypeParsers.java



```
} else if (propName.equals("analyzer") || // for backcompat, reading old indexes, remove for v3.0
        propName.equals("index_analyzer") && parserContext.indexVersionCreated().before(Version.V_2_0_0_beta1)) {

    NamedAnalyzer analyzer = parserContext.analysisService().analyzer(propNode.toString());
    if (analyzer == null) {
        throw new MapperParsingException("analyzer [" + propNode.toString() + "] not found for field [" + name + "]");
    }
    indexAnalyzer = analyzer;
    iterator.remove();
} else if (propName.equals("search_analyzer")) {
```

图 7.5 ES2.0 索引类型分词调整源代码

从中，我们可以明确看到前述作者的结论：ES V2.0.0.beta1 为一个分界线。

2. CRUD

（1）添加一条数据。

通过指定索引 Index 名、索引类型 Type 名和文档 ID，向索引库增加一条数据。

```
IndexResponse response = this.client.prepareIndex("test2",
            "tt2", "1").setSource(jsonBuilder(). startObject()
                                .field("name", "测试")
                                .field("gender", "boy")
                                .field("message", "es 测试插入")
                        .endObject()).get();
System.out.println(response.isCreated());//查看是否执行成功
```

在没有创建 Mapping 之前，上述添加索引代码执行后，ES 会自动创建好默认索引。

（2）查询一条数据：通过主键 ID 获取一条数据。

```
GetResponse response = this.client.prepareGet("test",
                "tt", "1").setOperationThreaded(false).get();
System.out.println(response.getSourceAsMap());
```

（3）删除一条数据：通过主键 ID 删除指定数据。

```
DeleteResponse response = this.client.prepareDelete("test",
                            "tt", "1").get();
System.out.println(response.isFound());
```

(4) 更新一条数据：通过主键 ID 修改指定字段信息，ES2.X 已经可以像关系 DB 一样通过 API 修改单个字段，ES1.X 修改某个字段，还只能通过覆盖更新整个文档。

```
UpdateResponse res = this.client.prepareUpdate("test", "tt", "1")
                    .setDoc(jsonBuilder().startObject()
                    .field("user", "王成光").endObject()).get();
System.out.println(res.getVersion());
```

修改单个字段一般根据主键 ID，类似于关系数据库的 update 操作。

7.4.3 ES 高级使用

1. 批量操作

批量操作有多种方式，像 MultiGetResponse、BulkRequestBuilder 和 BulkProcessor 等都可以一次性批量操作各种类型动作，但相比较而言，BulkProcessor 控制更精确。

```
BulkProcessorbulkProcessor = BulkProcessor.builder(this.client,
        new BulkProcessor.Listener() {
        @Override
        public   void   beforeBulk(long   executionId,BulkRequest
request) {
            System.out.println("buld request action num is: "
                        +request.numberOfActions());
        }

        @Override
        public   void   afterBulk(long   executionId,   BulkRequest
request,BulkResponse response) {
        System.out.println("buld request after, if has failure:"
                + response.hasFailures());
        }

        @Override
        public void afterBulk(long executionId,
            BulkRequest request, Throwable failure) {
        }
        })
            //execute the bulk every 10 000 requests
            .setBulkActions(10000)
            //flush the bulk every 1gb
            .setBulkSize(new ByteSizeValue(1, ByteSizeUnit.GB))
            .setFlushInterval(TimeValue.timeValueSeconds(5))
```

```
            //flush the bulk every 5 seconds whatever
            //the number of requests
            .setConcurrentRequests(1)
            //the number of concurrent requests. A value of 0 means
            //that only a single request will be allowed to be
            //executed. A value of 1 means 1 concurrent request is
            //allowed to be executed while accumulating new bulk
            //requests.
           .setBackoffPolicy(BackoffPolicy.exponentialBackoff (
                    TimeValue.timeValueMillis(100), 3)).build();
            String json = "{" +
                "\"user\":\"light\"," +
                "\"postDate\":\"2016-03-29\"," +
                "\"message\":\"中午买了一个平安穗\"" +
            "}";
        bulkProcessor.add(new IndexRequest(
                          "test", "tt", "5").source(json));
        bulkProcessor.add(new DeleteRequest("test", "tt", "4"));
        try {
            bulkProcessor.awaitClose(10, TimeUnit.MINUTES);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //awaitClose  close二者用其一即可
        //bulkProcessor.close();
```

批量操作构造器关闭时，方法 awaitClose() 和 close()二者只要选其一即可，如果两个都用，就会产生数据不能正常入库或删除等后果。这也是笔者经验之谈。

2. 多索引查找操作

同时从多个索引 Index 获取相关数据。这里提供两种方法。

方法 1：通过 MultiGet 方式。

```
    public void multiGetIndex(){
        //声明：MultiGet 响应对象：同时从索引 test 和 test2 中获取
        MultiGetResponse    multiGetItemResponses    =    this.client.
prepareMultiGet()
                .add("test", "tt", "1")
                .add("test", "tt", "2", "3", "AVPBEaZXr-Boc-pzZd2x")
                .add("test2", "tt2", "1")
                .add("test2", "tt2", "3").get();
        System.out.println(multiGetItemResponses.contextSize());
        String json = null;
        GetResponse response = null;
        //打印从两个索引中获取的每个 Document
        for(MultiGetItemResponse itemResponse:multiGetItemResponses) {
```

```
            response = itemResponse.getResponse();
            if (response.isExists()) {
        json = response.getSourceAsString();
        System.out.println(response.getIndex()+"\t"
                        +response.getType()+"\t"
                        +response.getId()+ "\t"+json);
            }
        }
    }
```

方法 2：通过 MultiSearch 方式。

```
public void multiSearch(){
    SearchRequestBuilder srb1 = this.client.prepareSearch("test")
                        .setQuery(QueryBuilders
                            .queryStringQuery ("es")).setSize(1);
    SearchRequestBuilder srb2 = this.client.prepareSearch("test2")
                        .setQuery(QueryBuilders
                            .matchQuery("message", "es")).setSize(1);
    //整合上述两个独立查询请求构造器：srb1 和 srb2
    MultiSearchResponse sr = this.client.prepareMultiSearch()
                        .add(srb1).add(srb2).execute().actionGet();
    long nbHits = 0;
    SearchHitsseh = null;
    for (MultiSearchResponse.Item item : sr.getResponses()) {
    SearchResponse response = item.getResponse();
    seh = response.getHits();
    nbHits += seh.getTotalHits();
        for(SearchHit sh : seh.getHits()){
        System.out.println("索引： " + sh.getIndex()
                            + " 索引 type: " + sh.getType()
                            + " id: " + sh.getId() + " 内容："
                            + sh.getSourceAsString());      }
    }
    System.out.println("查询结果总数："+nbHits);
}
```

上述两个方法，都可以同时从多个索引中获取所要结果，二者不同之处在于：方法 1 只能通过主键 ID 获取，而方法 2 可以指定搜索关键词。

3．聚合操作

参与聚合 AGG 操作的字段只索引不分词。

```
SearchResponse sr = this.client.prepareSearch()
                //设置查询关键词匹配
```

```
                .setQuery(QueryBuilders.matchAllQuery())
                //设置 AGG 相关字段
                .addAggregation(AggregationBuilders
                                .terms ("agg1").field("field"))
                //设置复杂的 AGG 字段，增加自定义字段范围聚合功能
                .addAggregation(
                AggregationBuilders
                        .dateRange("agg2").field("dateOfBirth")
                        .format("yyyy")
                        // from -infinity to 1950 (excluded)
                        .addUnboundedTo("1980")
                        // from 1950 to 1960 (excluded)
                        .addRange("1980", "1990")
                        .addUnboundedFrom("1990"))
                 .execute().actionGet();

// Get your facet results
Terms agg1 = sr.getAggregations().get("agg1");
for (Bucket b : agg1.getBuckets()) {
     System.out.println(b.getKey()+"\t"+b.getDocCount());
}

Range agg2 = sr.getAggregations().get("agg2");
String key= null;
DateTime fromAsDate=null, toAsDate=null;
long docCount = 0;
for (Range.Bucket entry : agg2.getBuckets()) {
        key = entry.getKeyAsString();   // Date range as key
        // Date bucket from as a Date
        fromAsDate = (DateTime) entry.getFrom();
        // Date bucket to as a Date
        toAsDate = (DateTime) entry.getTo();
        docCount = entry.getDocCount();  // Doc count
        System.out.println(key+"\t"
            +fromAsDate+"\t"+toAsDate+"\t"+docCount);
}
```

提供分组和统计文档的能力。聚合类似关系数据库中 group by 分组的功能，在 Elasticsearch 中，对一次的聚合查询中可以同时得到聚合的具体结果再次进行聚合，这是一个非常有用的功能。你可以通过一次操作得到多次聚合的结果，从而避免多次请求，减少网络和服务器的负担。

4. 集群管理操作

主要是方便 ES 集群基本信息即健康状况等检测等。

```
        //获取集群管理客户端
        ClusterAdminClientclusterAdminClient = this.client
                                              .admin(). cluster();
        //获取集群健康状态响应对象
        ClusterHealthResponse healths = clusterAdminClient
                                          . prepareHealth().get();
        String clusterName = healths.getClusterName();
        //获取集群 Data 节点个数
        int numberOfDataNodes = healths.getNumberOfDataNodes();
        //获取集群总的节点个数
        int numberOfNodes = healths.getNumberOfNodes();
        System.out.println(clusterName+"\t"
                        +numberOfDataNodes+"\t"+numberOfNodes);

        String index = null;
        int numberOfShards=-1,numberOfReplicas=-1;
        ClusterHealthStatus status = null;
    //依次列举集群中每个索引的简况状态
        for (ClusterIndexHealth health
                            : healths.getIndices(). values()) {
            index = health.getIndex();
            //获取当前索引 index 的分片数
            numberOfShards = health.getNumberOfShards();
            //获取当前索引 index 的副本数
            numberOfReplicas = health.getNumberOfReplicas();
            //获取当前索引 index 的健康状态：绿色：正常，黄色：警告，红色：故障。
            status = health.getStatus();
            System.out.println(index+"\t"+numberOfShards+"\t"
                            + numberOfReplicas+"\t"+status);
        }
```

集群管理，主要是便于 ES 运维管理人员精确掌控集群运行状况，避免生产环境下故障不能及时处理，从而影响线上服务。

7.5 ES 在自研框架中的作用

自研框架旨在解决当今大数据时代的以用户行为为基础的用户画像实时更新类似问题，这里的用户行为是指用户所浏览、收藏、分享、购买及搜索行为信息，所涉及的对象主要是新闻、博客、视频、广告及商品等，这些元数据索引信息比较适合存在 ES 中，此外，用户的长期兴趣标签，本身信息维度也比较多，再考虑时间因素，也是需要多维度查找，因此也比较适合存在 ES 中。

为此，我们可以在 ES 中设置各类元数据索引和用户兴趣标签索引，既涉及

索引信息的更新（整条元数据更新或某几个个字段的更新），同时也有大量查询请求处理，这里我们统一使用 ES 索引更新 API，或索引查询 Query 相关 API。

因为 ES 针对索引查询所提供基于 Web Service 的 Restful API 使用非常方便，只要在浏览器地址栏输入：

http://${ES 集群域名:端口}/${索引名}/${索引类型}/${索引主键 ID}

就可以得到相关索引的某个文档详细信息。此外，为查找方便和性能考虑，也可以通过 ES 提供的基于 TCP 的 TransportClient 连接服务，也是生产环境下应用程序默认使用方式。

通常用得比较多的查询有两类函数，分别是

- 基于 ID 集合的查询函数：QueryBuilders.idsQuery().addIds(idList)。
- 关键词多字段不同权重检索：例如查询关键词 kword 使用分词器“ik”在字段 title、digest、dkwords 和 content 中检索。代码如下：

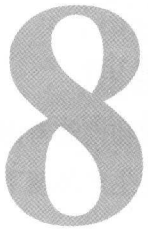
```
QueryBuilders.queryStringQuery(kword)
                .analyzer("ik").field("title", 3) .field("digest", 2)
                .field("dkwords", 2).field("content");
```

- 基于时间范围过滤函数：例如查询最近晚上 6 点半到晚 8 点半时间段的信息，代码结构如下：QueryBuilders.rangeQuery("addTime")
  .gte(20160331182835L).lt(20160331202835L)。

这里 gte()表示“>=”，lt()表示“<”，二者结合表示一个时间段。

7.6 总结

本章首先向读者对比介绍了搜索 ES 和 Solr 的特性，经实验对比，ES 总体比 Solr 更适合生产环境使用。接着重点向读者介绍了 ES 集群的几个核心步骤，以及相关原理和注意事项，再次向读者说明了 ES 客户端 Java 语言开发的基本操作，以及高级使用示例，希望通过阅读本章，让读者可以对全文检索有一个全新认识，并可以快速上手。



第 8 章 微服务架构通信——RPC 和 Web Service

微服务架构模式（Microservice Architect Pattern）近几年在软件架构模式领域广为流行。微服务间的通信就是 IPC（inter process communication），已有很多成熟方案，现在通用的技实现方式主要有两种：异步消息调用和同步调用，其中异步消息调用，第 4 章已经专门介绍，这里主要向读者朋友介绍同步调用，同步调用当前又主要分为两种模式：

1）RPC（Thrift、Avro、Dubbo、GRPC、Zero-ICE），传输协议高效，安全可控，特别在公司内部，如果有统一的开发规范及服务框架，其开发效率优势更明显些。当前众多 RPC 框架，本身支持常用的各种开发语言，跨语言开发不是问题。

2）REST（JWS、Jetty、SpringMVC），一般基于 HTTP 的 Web Service，更容易实现、被接受，服务端实现技术也更灵活些，各个语言都能支持，对客户端没有特殊的

8.1 微服务架构由来

以笔者自身从事软件开发十余年的经验，服务开发模式先后经历了：传统的

单体式架构、重量级 ESB2.0/ESB3.0、基于 Axis2 实现 WSDL 的 Web Service、以 SpringMVC 为代表的轻量级 RestFul 风格 Web Service、以 Thrift / Dubbo 为代表的 RPC 等。就单体式架构开发而言，以广为使用的 Web 开发为例，也先后经历了由 Struts、Spring 和 Hibernate 及展示层的 Velocity 转变为当前以 SpringMVC、SpringData（整合常用 NoSQL：Mongo、Redis、ES 和 Solr 等）、MyBatis（主要是关系 DB）和 Freemarker 的 Web 开发框架，可以说，每次技术实现方式的调整，都代表软件行业的一次进步。

从业界的讨论来看，微服务本身并没有一个严格的定义，这里我们可以借鉴业界大拿 ThoughtWorks 的首席科学家，马丁 · 福勒先生对微服务的这段描述[1]，似乎更加具体、贴切，通俗易懂：微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通（通常是基于 HTTP 协议的 RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

8.1.1 微服务与 SOA 比较

SOA（Service-Oriented Architecture）面向服务的体系架构，实际上是一个组件模型，它将应用程序的不同功能单（称为服务）通过定义良好的接口联系起来。SOA 建立在 Web 服务的基础之上，可以看做是 B/S 模型、XML/Web Service 技术之后的自然延伸。它根据需求通过网络松散耦合的粗粒度应用组件进行分布式部署、组合和使用。SOA 的核心是“服务”，本质就是服务组合起来对外提供接口。

微服务和 SOA 是一脉相承的，大神 Martin Fowlert 所提出的微服务可以说是 SOA 的理念继续升华，精进了一步。其核心思想是在应用开发领域，使用一系列

1 微服务架构综述 http://www.infoq.com/cn/articles/analysis-the-architecture-of-microservice-part-02

微小服务来实现单个应用的方式途径，或者说微服务的目的是有效地拆分应用，实现敏捷开发和部署，可以使用不同的编程语言编写。而 SOA 可能包含的意义更泛一些，更不准确一些。二者之间一个最本质的区别就在于“Smart endpoints and dumb pipes”，或者说是真正的分布式的、去中心化的。“Smart endpoints and dumb pipes”本质就是去 ESB，把所有“思考逻辑包括路由、消息解析等放在服务内部（Smart endpoints），去掉一个大一统的 ESB，服务间轻（dumb pipes）通信，是比 SOA 更彻底的拆分。

表 8.1 给出了 SOA 实现和微服务架构实现的区别的基本描述。

表 8.1 SOA 实现和微服务架构实现的区别

SOA 实现	微服务架构实现
企业级，自顶向下开展实施	团队级，自底向上开展实施
服务由多个子系统组成，粒度大	一个系统被拆分成多个服务，粒度细
企业服务总线，集中式的服务架构	无集中式总线，松散的服务架构
集成方式复杂（ESB/WS/SOAP）	集成方式简单（HTTP/REST/JSON）
单块架构系统，相互依赖，部署复杂	服务都能独立部署

SOA 的提出是在企业计算领域，目的是要将紧耦合的系统，划分为面向业务的、粗粒度、松耦合、无状态的服务。服务发布出来供其他服务调用，一组互相依赖的服务就构成了 SOA 架构下的系统。而企业计算领域，如果不是交易系统的话，并发量都不是很大的，所以大多数情况下，一台服务器就容纳将许许多多的服务，这些服务采用统一的基础设施，可能都运行在一个应用服务器的进程中。虽然说是面向服务了，但还是单一的系统。

微服务架构大体是从互联网企业兴起的，由于大规模用户，对分布式系统的要求很高，如果像企业计算那样的系统，伸缩就需要多个容纳续续多多的服务的系统实例，前面通过负载均衡使得多个系统成为一个集群。微服务与 SOA 相比，更强调分布式系统的特性，比如横向伸缩性、服务发现、负载均衡、故障转移、高可用。互联网开发对服务治理提出了更多的要求，比如多版本，比如灰度升级，比如服务降级，比如分布式跟踪，这些都是在 SOA 实践中重视不够的。此时自然倾向采用以子系统为分割，不同的子系统采用自己的架构，那么各个服务运行自己的 Web 容器中，当要增加计算能力的时候，只需要增加这个子系统或服务的实

例就好了，当升级的时候，可以不影响别的子系统。这种组织方式大体上就被称作微服务架构。

相比传统 SOA 的服务实现方式，微服务更具有灵活性、可实施性，以及可扩展性，其强调的是一种独立测试、独立部署、独立运行的软件架构模式。Docker 容器技术的出现，为微服务提供了更便利的条件，比如更小的部署单元，每个服务可以通过类似 Node.js 或 Spring Boot 的技术跑在自己的进程中。可能在几十台计算机中运行成千上万个 Docker 容器，每个容器都运行着服务的一个实例。随时可以增加某个服务的实例数，或者某个实例崩溃后，在其他的计算机上再创建该服务的新的实例。

8.1.2 微服务架构的优缺点

1. 微服务优点

(1) 复杂度可控：它将一个可怕的、庞大的整体应用分解成一组服务，在整体的功能没有改变的同时，应用程序已经被分解成可管理的模块或服务。每一个微服务专注于单一功能，并通过定义良好的接口以 RPC 或者消息驱动 API 形式定义清楚的界限。由于体积小、复杂度低，每个微服务可由一个小规模开发团队完全掌控，易于保持高可维护性和开发效率。

(2) 独立部署：由于微服务具备独立的运行进程，所以每个微服务也可以独立部署。开发者再也不需要调整本地对其服务的变更而进行部署。当某个微服务发生变更时无需编译、部署整个应用。由微服务组成的应用相当于具备一系列可并行的发布流程，使得发布更加高效，同时降低对生产环境所造成的风险，最终缩短应用交付周期。

(3) 技术选型灵活：微服务架构下，技术选型是去中心化的。每个团队可以根据自身服务的需求和行业发展的现状，自由选择最适合的技术栈，这种自由意味着开发者不再受限于使用可能过时的技术开始新的项目。由于每个微服务相对简单，当需要对技术栈进行升级时所面临的风险较低，甚至完全重构一个微服务也是可行的。

(4) 扩展：单块架构应用也可以实现横向扩展，就是将整个应用完整地复制到不同的节点。当应用的不同组件在扩展需求上存在差异时，微服务架构便体现出其灵活性，因为每个服务可以根据实际需求独立进行扩展。

(5) 每个微服务都有自己的存储能力，可以有自己的数据库，也可以有统一数据库。

(6) 微服务易于被一个开发人员理解、修改和维护，这样小团队能够更关注自己的工作成果，无需整体协作就能体现价值。

(7) 微服务是松耦合的，是有功能意义的服务，无论是在开发阶段或部署阶段都是独立的。

2. 微服务缺点

(1) 过分强调服务的规模：实际上有些开发者支持构建极其细粒度的服务，尽管规模小的服务更可取，但是最好记住这只是手段而不是目的。微服务的目的是充分地分解应用程序，以促进敏捷开发和部署。

(2) 为分布式系统带来的复杂性：开发者需要选择或者实现基于消息或 RPC 的进程间通信机制，而且必须编写处理部分失败的代码，因为请求的目的地可能很慢或者不可用。虽然这都不是高深莫测的事情，但是相对于整体应用程序，这明显更复杂，因为整体应用程序中模块间的调用是通过语言层面的方法/程序调用实现的。

(3) 分割的数据库架构：更新多个业务实体的事务相当普遍，这种事务在整体应用程序中很容易实现，因为只有一个数据库。而在基于微服务的应用中，你需要更新多个属于不同服务的数据库。分布式事务通常不是最好的选择，不仅仅因为 CAP 理论，而且目前许多高扩展性的 NoSQL 数据库和消息代理就不支持。你最终不得不使用基于最终一致性的方法，这对于开发者来说更具挑战性。

(4) 测试微服务也很复杂：使用 Spring Boot 这样的现代框架很容易开始开发一个整体 Web 应用程序，编写测试类测试其 REST API。与此相反，对于微服务

的一个类似的测试则需要运行该服务，以及依赖的服务。这也不是高深莫测的事情，但是不要低估做这些事情的复杂性。

(5) 实现跨服务的需求变更很复杂：设想你实现的用户故事需要更改服务 A，因为 A 依赖 B，B 依赖 C，所以你又得更改服务 B 和 C。在微服务架构模式中，你需要仔细计划和协调各个服务的更改上线。例如你需要首先更新服务 C，接着是服务 B，最后才是服务 A。很幸运的是大部分改动通常只影响一个服务，需要协调的跨服务更改相对较少。

(6) 部署一个基于微服务的应用也很复杂：微服务应用通常包含大量服务。每个服务有多个运行实例。有太多运行的实例需要配置、部署、扩展和监控。另外也需要实现一个服务发现机制，以确保某服务可以找到需要通信的其他服务的位置。因此成功的部署微服务应用需要开发者对部署方法有更强的控制，已经更高水平的自动化。

8.1.3 微服务雪崩效应的防范

. 何谓雪崩效应

由于微服务间通过 RPC 来进行数据交换，所以我们可以做一个假设：在 IO 型服务中，假设服务 A 依赖服务 B 和服务 C，而 B 服务和 C 服务有可能继续依赖其他的服务，继续下去会使得调用链路过长，技术上称 1->N 扇出。如果在 A 的链路上某个或几个被调用的子服务不可用或延迟较高，则会导致调用 A 服务的请求被堵住，堵住的请求会消耗占用掉系统的线程、I/O 等资源，当该类请求越来越多，占用的计算机资源越来越多的时候，会导致系统瓶颈出现，造成其他的请求同样不可用，最终导致业务系统崩溃，又称：雪崩效应[2]。

一般情况对于服务依赖的保护，解决或缓解服务雪崩的方案主要有 3 种：

(1) 熔断模式：这种模式主要是参考电路熔断，如果一条线路电压过高，保

2 微服务熔断与隔离 http://yq.aliyun.com/articles/7443

险丝会熔断，防止火灾。放到我们的系统中，如果某个目标服务调用慢或者有大量超时，此时，熔断该服务的调用，对于后续调用请求，不再继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。熔断的设计主要参考了 hystrix（Hystrix 官方文档：https://github.com/Netflix/Hystrix/wiki）的做法。其中最重要的是三个模块：熔断请求判断算法、熔断恢复机制、熔断报警。

- **熔断请求判断算法**：使用无锁循环队列计数，每个熔断器默认维护 10 个 bucket，每 1 秒一个 bucket，每个 bucket 记录请求的成功、失败、超时、拒绝的状态，默认错误超过 50%且 10 秒内超过 20 个请求进行中断拦截。
- **熔断恢复机制**：对于被熔断的请求，每隔 5s 允许部分请求通过，若请求都是健康的（RT<250ms）则对请求健康恢复。
- **熔断报警**：对于熔断的请求找日志，异常请求超过某些设定则报警。

（2）隔离模式：这种模式就像对系统请求按类型划分成一个个小岛一样，当某个小岛被火烧光了，不会影响到其他的小岛。例如可以对不同类型的请求使用线程池来资源隔离，每种类型的请求互不影响，如果一种类型的请求线程资源耗尽，则对后续的该类型请求直接返回，不再调用后续资源。这种模式使用场景非常多，例如将一个服务拆开，对于重要的服务使用单独服务器来部署，再或者最近推广的多中心。隔离的方式一般使用两种:

- **线程池隔离模式**：使用一个线程池来存储当前的请求，线程池对请求进行处理，设置任务返回处理超时时间，请求堆积入线程池队列。这种方式需要为每个依赖的服务申请线程池，有一定的资源消耗，好处是可以应对突发流量(流量洪峰来临时,处理不完可将数据存储到线程池队里慢慢处理)。
- **信号量隔离模式**：使用一个原子计数器（或信号量）来记录当前有多少个线程在运行，请求先判断计数器的数值，若超过设置的最大线程个数则丢弃改类型的新请求，若不超过则执行计数操作请求来计数器+1，请求返回计数器-1。这种方式是严格控制线程且立即返回模式，无法应对突发流量（流量洪峰来临时，处理的线程超过数量，其他的请求会直接返回，不会继续去请求依赖的服务）

（3）限流模式：上述的熔断模式和隔离模式都属于出错后的容错处理机制，

而限流模式则可以称为预防模式。限流模式主要是提前对各个类型的请求设置最高的 QPS 阈值，若高于设置的阈值则对该请求直接返回，不再调用后续资源。这种模式不能解决服务依赖的问题，只能解决系统整体资源分配问题，因为没有被限流的请求依然有可能造成雪崩效应。超时机制设计主要分两种：

- **等待超时**：在任务入队列时设置任务入队列时间，并判断队头的任务入队列时间是否大于超时时间，超过则丢弃任务。
- **运行超时**：直接可使用线程池提供的 get 方法。

8.2 RPC 介绍及实践

目前 RPC 服务有很多，这里重点给读者介绍几种目前业界广为使用的开源 RPC 框架。

8.2.1 Thrift/Nifty 介绍

1. 基本认识

Thrift 是一个远程过程调用（RPC）软件框架，用来进行可扩展且跨语言的服务的开发。它结合了功能强大的软件堆栈和代码生成引擎，以构建在 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk 和 OCaml 等这些编程语言间无缝结合的高效服务。Thrift 最初由 Facebook 开发，2007 年 4 月开放源码，2008 年 5 月进入 Apache 孵化器，现在是 Apache 基金会的顶级项目。Thrift 允许你在一个简单的定义文件中定义数据类型和服务接口，以作为输入文件，编译器生成代码用来方便地生成 RPC 客户端和服务器通信的无缝跨编程语言。目前官方最新版是 Version0.9.3。

Nifty 是 Facebook 公司开源的，基于 Netty 的 Thrift 服务端和客户端实现，使用此包就可以快速发布出基于 Netty 的高效的服务端和客户端代码。目前官方最新版 Version0.18.0，其使用方式和 Thrift 大致相同，不同之处仅仅在服务启动时 API，底层实现一致，这里就不过多介绍。

著名的 Key-Value 存储服务器 Cassandra 就是使用 Thrift 作为其客户端 API，分布式实时计算框架 Storm 采用 Thrift 作为底层通信框架。

2. 技术要领

如图 8.1 所示，图中**黄色**部分是用户自定义实现的业务逻辑[3]，**褐色**部分是根据 Thrift 定义的服务接口描述文件（IDL）生成的客户端和服务器端代码框架，**红色**部分是根据 Thrift 文件生成代码实现数据的读写操作。红色部分以下是 Thrift 的传输体系、协议，以及底层 I/O 通信，也即 Thrift 的三个主要的组件：**Protocol**，**Transport** 和 **Server**，其中，Protocol 定义了消息是怎样序列化的，Transport 定义了消息是怎样在客户端和服务器端之间通信的，Server 用于从 Transport 接收序列化的消息，根据 Protocol 反序列化之，调用用户定义的消息处理器，并序列化消息处理器的响应，然后再将它们写回 Transport。

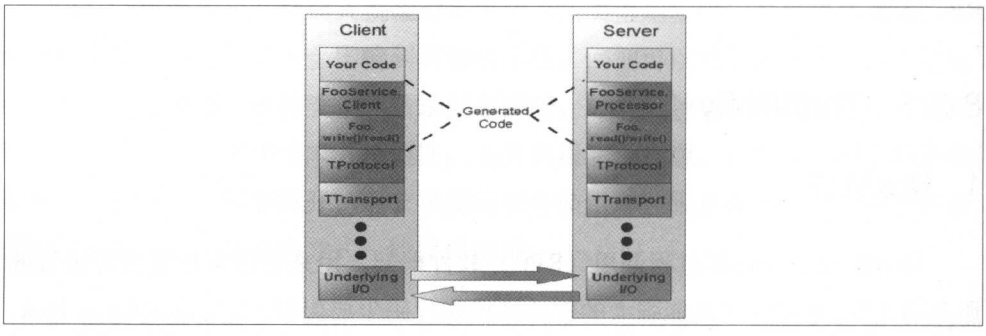


图 8.1 Thrift 架构

使用 Thrift 可以很方便地定义一个服务并且选择不同的传输协议和传输层而不用重新生成代码。Thrift 服务器包含用于绑定协议和传输层的基础架构，它提供阻塞、非阻塞、单线程和多线程的模式独立运行在服务器上，也可以配合服务器/容器一起运行，和现有的 J2EE 服务器/Web 容器无缝结合。

（1）**协议**：Thrift 可以让用户选择客户端与服务端之间传输通信协议的类别，在传输协议上总体划分为文本（text）和二进制（binary）传输协议，为节约带宽，

3 Thrift 应用总结 http://www.tuicool.com/articles/rq6nMv

提高传输效率，一般情况下使用二进制类型的传输协议为多数，根据项目/产品中的实际需求，有时还会使用基于文本类型的协议。常用协议有以下几种：

- TBinaryProtocol——二进制编码格式进行数据传输。
- TCompactProtocol——高效率的、密集的二进制编码格式进行数据传输。
- TJSONProtocol——使用 JSON 的数据编码协议进行数据传输。
- TSimpleJSONProtocol——只提供 JSON 只写的协议，适用于通过脚本语言解析。

（2）**传输层**：常用的传输层有以下几种：

- TSocket——使用阻塞式 I/O 进行传输，是最常见的模式。TFramedTransport——使用非阻塞方式，按块的大小进行传输，类似于 Java 中的 NIO。若使用 TFramedTransport 传输层，其服务器必须修改为非阻塞的服务类型。
- TNonblockingTransport——使用非阻塞方式，用于构建异步客户端。

（3）**Server 层**：Thrift 目前提供的几种 Server 处理方式如下。

- **TSimplerServer**：接收一个连接，处理连接请求，直到客户端关闭了连接，它才回去接收一个新的连接。正因为它只在一个单独的线程中以阻塞 I/O 的方式完成这些工作，所以它只能服务一个客户端连接，其他所有客户端在被服务器端接收之前都只能等待。TSimpleServer 主要用于测试目的，不要在生产环境中使用它！
- **TNonblockingServer**：使用非阻塞的 I/O 解决了 TSimpleServer 一个客户端阻塞其他所有客户端的问题。它使用了 java.nio.channels.Selector，通过调用 select()，它使得你阻塞在多个连接上，而不是阻塞在单一的连接上。当一或多个连接准备好被接受读/写时，select()调用便会返回。TNonblockingServer 处理这些连接的时候，要么接收它，要么从它那读取数据，要么把数据写到它那里，然后再次调用 select()来等待下一个可用的连接。通用这种方式，Server 可同时服务多个客户端，而不会出现一个客户端把其他客户端全部“饿死”的情况。然而，这样会有个棘手的问题：所有消息是被调用 select()方法的同一个线程处理的。假设有 10 个客户端，

处理每条消息所需时间为 100 毫秒，那么，latency 和吞吐量分别是多少？当一条消息被处理的时候，其他 9 个客户端就等着被 select，所以客户端需要等待 1 秒钟才能从服务器端得到回应，吞吐量就是 10 个请求/秒。如果可以同时处理多条消息的话，会很不错吧？

- **THsHaServer**：半同步/半异步的 server 就应运而生了。它使用一个单独的线程来处理网络 I/O，一个独立的 worker 线程池来处理消息。这样，只要有空闲的 worker 线程，消息就会被立即处理，因此多条消息能被并行处理。用上面的例子来说，现在的 latency 就是 100 毫秒，而吞吐量就是 100 个请求/秒。THsHaServer 能够并行处理所有请求，而 TNonblockingServer 只能一次处理一个请求。
- **TThreadedSelectorServer**：它与 THsHaServer 的主要区别在于，TThreadedSelectorServer 允许你用多个线程来处理网络 I/O。它维护了两个线程池：一个用来处理网络 I/O，另一个用来进行请求的处理。当网络 I/O 是瓶颈的时候，TThreadedSelectorServer 比 THsHaServer 的表现要好。为了展现它们的区别，我进行了一个测试，令其消息处理器在不做任何工作的情况下立即返回，以衡量在不同客户端数量的情况下的平均 latency 和吞吐量。对 THsHaServer，我使用 32 个 worker 线程；对 TThreadedSelectorServer，我使用 16 个 worker 线程和 16 个 selector 线程。TThreadedSelectorServer 比 THsHaServer 的吞吐量高得多，并且维持在一个更低的 latency 上。
- **TThreadPoolServer**：有一个专用的线程用来接收连接，一旦接收了一个连接，它就会被放入 ThreadPoolExecutor 中的一个 worker 线程里处理。worker 线程被绑定到特定的客户端连接上，直到它关闭。一旦连接关闭，该 worker 线程就又回到了线程池中。你可以配置线程池的最小、最大线程数，默认值分别是 5（最小）和 Integer.MAX_VALUE（最大）。如果客户端数量超过了线程池中的最大线程数，在有一个 worker 线程可用之前，请求将被一直阻塞在那里。TThreadPoolServer 的表现非常优异。在我们正在使用的服务器上，它可以支持 1 万个并发连接而没有任何问题。如果你提前知道了将要连接到你服务器上的客户端数量，并且你不介意运行大量线程的话，TThreadPoolServer 对你可能是个很好的选择。

(4) 数据类型：如表 8.2 所示，展示了 Thrift 目前支持的数据类型。

表 8.2 Thrift 数据类型说明

数据类型	详细说明
基本类型	bool：布尔值，true 或 false，对应 Java 的 boolean byte： 8 位有符号整数，对应 Java 的 byte i16： 16 位有符号整数，对应 Java 的 short i32： 32 位有符号整数，对应 Java 的 int i64： 64 位有符号整数，对应 Java 的 long double：64 位浮点数，对应 Java 的 double string：未知编码文本或二进制字符串，对应 Java 的 String
结构体类型	struct：定义公共的对象，类似于 C 语言中的结构体定义，在 Java 中是一个 JavaBean
容器类型	list：对应 Java 的 ArrayList set：对应 Java 的 HashSet map：对应 Java 的 HashMap
异常类型	exception：对应 Java 的 Exception
服务类型	service：对应服务的类

3. Thrift的使用

Thrift 安装部署相对比较麻烦，但我们可以使用其提供 Windows 版的可执行 exe 文件，执行相应 IDL，快速生成各个语言版的客户端代码。笔者自研框架分布式实时计算框架 light_drtc 就是使用的 Thrift 最新版 0.9.3 生成的相应客户端代码。一般使用方式：

```
thrift --gen <language> <Thrift filename>
```

使用实例，可以参考 light_drtc 源代码部分。使用时，需要借助 Maven 引入 Jar：

```
<dependency>
    <groupId>org.apache.thrift</groupId>
    <artifactId>libfb303</artifactId>
    <version>0.9.3</version>
</dependency>
```

将事先定义好的 IDL 生成相关客户端 API，待服务端启动时，即可完成服务端和客户端 RPC 通信。

8.2.2 Avro 介绍

1. 基本认识

Avro 是 Hadoop 的一个子项目，由 Hadoop 的创始人 Doug Cutting（也是 Lucene，Nutch 等项目的创始人）牵头开发。Avro 是一个基于二进制数据传输高性能的中间件[4]。在 Hadoop 的其他项目中例如 HBase(Ref)和 Hive(Ref)的 Client 端与服务端的数据传输也采用了这个工具。Avro 是一个数据序列化的系统。Avro 可以将数据结构或对象转化成便于存储或传输的格式。Avro 设计之初就用来支持数据密集型应用，适合于远程或本地大规模数据的存储和交换。它的主要特点有：

- 丰富的数据结构类型；
- 快速可压缩的二进制数据形式，对数据二进制序列化后可以节约数据存储空间和网络传输带宽；
- 存储持久数据的文件容器；
- 可以实现远程过程调用 RPC；
- 简单的动态语言结合功能。

当前市场上有很多类似的序列化系统，如 Google 的 Protocol Buffers, Facebook 的 Thrift。Avro 提供着与诸如 Thrift 和 Protocol Buffers 等系统相似的功能，但是在一些基础方面还是有区别的，主要是：

- 动态类型：Avro 并不需要生成代码，模式和数据存放在一起，而模式使得整个数据的处理过程并不生成代码、静态数据类型等等。这方便了数据处理系统和语言的构造。
- 未标记的数据：由于读取数据的时候模式是已知的，那么需要和数据一起编码的类型信息就很少了，这样序列化的规模也就小了。
- 不需要用户指定字段号：即使模式改变，处理数据时新旧模式都是已知的，所以通过使用字段名称可以解决差异问题。

Avro 依赖模式（Schema）来实现数据结构定义，只有确定了模式才能对数据

4 Avro 简介 http://www.open-open.com/lib/view/open1369363962228.html

进行解释，所以在数据的序列化和反序列化之前，必须先确定模式的结构。正是模式的引入，使得数据具有了自描述的功能，同时能够实现动态加载，可以把模式理解为 Java 的类，它定义每个实例的结构，可以包含哪些属性，根据类来产生任意多个实例对象。对实例序列化操作时必须需要知道它的基本结构，也就需要参考类的信息。

所以，在 Avro 可用的一些场景下，如文件存储或是网络通信，都需要模式与数据同时存在。Avro 数据以模式来读和写（文件或是网络），并且写入的数据都不需要加入其他标识，这样序列化时速度快且结果内容少。由于程序可以直接根据模式来处理数据，所以 Avro 更适合于脚本语言的发挥。Avro 官方网址为 https://avro.apache.org/，最新版是 2016 年 1 月 30 日发布的 V1.8.0。

2. 技术要领

（1）数据类型

数据类型标准化的意义是：一方面使不同系统对相同的数据能够正确解析，另一方面，数据类型的标准定义有利于数据序列化/反序列化。Avro 数据类型分为两类：基础数据类型和复杂数据类型。

基础数据类型，Avro 定义了 8 种基础数据类型，表 8.3 是其简单说明。

表 8.3 Avro 基础数据类型说明

类　型	说　明
null	表示没有值 0 字节
boolean	表示一个二进制布尔值，一个字节：0-false，1-true
int	表示 32 位有符号整数
long	表示 64 位有符号整数
float	表示 32 位的单精度浮点数（IEEE 754）4 字节
double	表示 64 位的单精度浮点数（IEEE 754）8 字节
bytes	表示 8 位的无符号字节序列
string	Unicode 编码的字符序列

基础数据类型由类型名称定义，不包含属性信息，例如字符串定义如下：{"type": "string" }。基础数据类型也可以使用 JSON 定义类型名称，比如 schema

"string"和{"type": "string"}是同义且相等的。

复杂数据类型：Avro 定义了 6 种复杂数据类型，每一种复杂数据类型都具有独特的属性，表 8.4 就每一种复杂数据类型进行说明。每一种复杂数据类型都含有各自的一些属性，其中部分属性是必需的，部分是可选的。

表 8.4 Avro 复杂数据类型说明

类型	属性	说　明
Records	**type**	record
	name	是一个 JSON string，提供了记录的名字，为必有属性
	namespace	是一个 JSON string，用来限定和修饰 name 属性，动态方式生成后，为 Java 的包名，为可选属性
	doc	是 JSON string，为使用这个 Schema 的用户提供文档，可选属性
	aliases	是 JSON 的一个 string 数组，为这条记录提供别名，可选属性
	fields	必选属性，是一个 JSON 数组，数组中列举了所有的 field。每一个 field 都是一个 JSON 对象，并且具有如下属性：
	name	必选属性，field 的名字，是一个 JSON string，类似一个 Java 属性。
	doc	可选属性，为使用此 Schema 的用户提供了描述此 field 的文档
	type	必选属性，定义 Schema 的一个 JSON 对象，或者是命名一条记录定义的 JSON string。
	default	可选属性，即 field 的默认值，当读到缺少这个 field 的实例时用到
Enums	**type**	**enum**
	name	必有属性，是一个 JSON string，提供了 enum 的名字
	namespace	也是一个 JSON string，用来限定和修饰 name 属性
	doc	可选属性，是一个 JSON string，为使用这个 Schema 的用户提供文档
	aliases	可选属性，是 JSON 的一个 string 数组，为这个 enum 提供别名
	symbols	必有属性，是一个 JSON string 数组，列举了所有的 symbol，在 enum 中的所有 symbol 都必须是唯一的，不允许重复
Arrays	**type**	**array**
	items	array 中元素的 Schema
Maps	**type**	**map**
	values	用来定义 map 的值的 Schema
Fixed	**type**	**fixed**
	name	为这个 fixed 命名的一个字符串（**required**）.
	namespace	a string that qualifies the name.
	aliases	是 JSON 的一个 string 数组，为这个 enum 提供别名（optional）.
	size	每个值的字节个数（**required**）.
Unions		JSON 数组

（2）序列化/反序列化

Avro 支持两种序列化编码方式：二进制编码和 JSON 编码。使用二进制编码会高效序列化，并且序列化后得到的结果会比较小；而 JSON 一般用于调试系统或是基于 Web 的应用。对 Avro 数据序列化/反序列化时都需要对模式以深度优先（Depth-First），从左到右（Left-to-Right）的遍历顺序来执行。基本类型的序列化容易解决，混合类型的序列化会有很多不同规则。对于基本类型和混合类型的二进制编码在文档中规定，按照模式的解析顺序依次排列字节。对于 JSON 编码，联合类型（Union Type）就与其他混合类型表现不一致。

图 8.2 表示的是 Avro 本地序列化和反序列化的实例，它将用户定义的模式和具体的数据编码成二进制序列存储在对象容器文件中，例如用户定义了包含学号、姓名、院系和电话的学生模式，而 Avro 对其进行编码后存储在 student.db 文件中，其存储数据的模式放在文件头的元数据中，这样读取的模式即使与写入的模式不同，也可以迅速地读出数据。假如另一个程序需要获取学生的姓名和电话，只需要定义包含姓名和电话的学生模式，然后用此模式去读取容器文件中的数据即可。

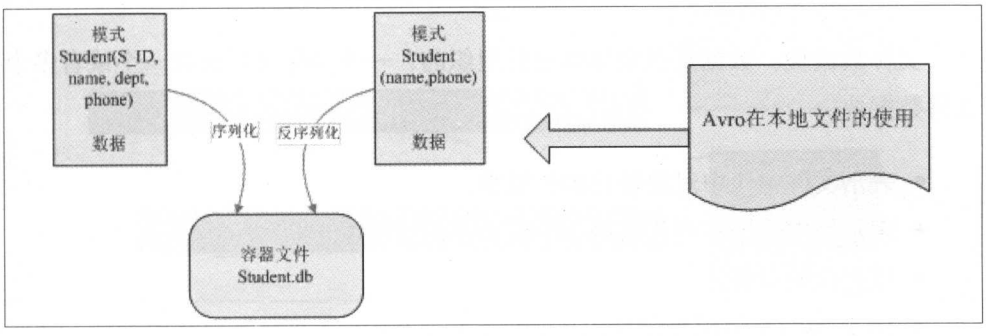


图 8.2　Avro 序列化实例

在 Avro 进行 RPC 通信时，我们可以有两种方式来实现：一种是静态方式，采用 Protocol 文件来生成所需要的类，然后直接调用类里面的实现；另一种是动态方式，直接采用代码解析 Protocol 文件内容，动态设置内容。

（3）容器文件

Avro 定义了一个简单的对象容器文件格式。一个文件对应一个模式，所有存

储在文件中的对象都是根据模式写入的。对象按照块进行存储，块可以采用压缩的方式存储。为了在进行 mapreduce 处理的时候有效地切分文件，在块之间采用了同步记号。一个文件可以包含任意用户定义的元数据。

一个文件由两部分组成：文件头和一个或者多个文件数据块。

文件头：

- 四个字节，ASCII ‘O’，‘b’，‘j’，1。
- 文件元数据，用于描述 Schema。
- 16 字节的文件同步记号。
- 其中，文件元数据的格式为：
  - 值为-1 的长整型，表明这是一个元数据块。
  - 标识块长度的长整型。
  - 标识块中 key/value 对数目的长整型。
  - 每一个 key/value 对的 string key 和 bytesvalue。
  - 标识块中字节总数的 4 字节长的整数。

文件数据块：数据是以块结构进行组织的，一个文件可以包含一个或者多个文件数据块。

- 表示文件中块中对象数目的长整型。
- 表示块中数据序列化后的字节数长度的长整型。
- 序列化的对象。
- 16 字节的文件同步记号。

当数据块的长度为 0 时即为文件数据块的最后一个数据，此后的所有数据被自动忽略。

如图 8.3 所示，一个存储文件由两部分组成：头信息（Header）和数据块（Data Block）。而头信息又由三部分构成：四个字节的前缀、文件 Meta-data 信息和随机生成的 16 字节同步标记符。Avro 目前支持的 Meta-data 有两种：schema 和 codec。codec 表示对后面的文件数据块（File Data Block）采用何种压缩方式。Avro 的实

现都需要支持下面两种压缩方式：null（不压缩）和 deflate（使用 Deflate 算法压缩数据块）。除了文档中认定的两种 Meta-data，用户还可以自定义适用于自己的 Meta-data。这里用 long 型来表示有多少个 Meta-data 数据对，也是让用户在实际应用中可以定义足够的 Meta-data 信息。对于每对 Meta-data 信息，都有一个 string 型的 key（需要以“avro.”为前缀）和二进制编码后的 value。对于文件中头信息之后的每个数据块，有这样的结构：一个 long 值记录当前块有多少个对象，一个 long 值用于记录当前块经过压缩后的字节数，真正的序列化对象和 16 字节长度的同步标记符。由于对象可以组织成不同的块，使用时就可以不经过反序列化而对某个数据块进行操作。还可以由数据块数、对象数和同步标记符来定位损坏的块以确保数据完整性。

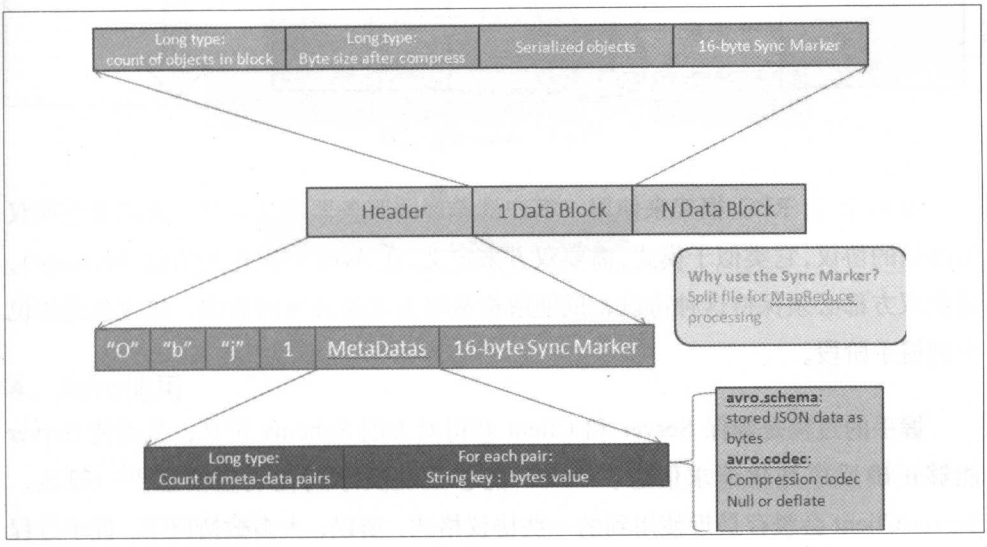


图 8.3 Avro 容器文件

3. RPC实现

当在 RPC 中使用 Avro 时，服务器和客户端可以在握手连接时交换模式。服务器和客户端有彼此全部的模式，因此相同命名字段、缺失字段和多余字段等信息之间通信中需要处理的一致性问题就可以容易解决。如图 8.4 所示，协议中定义了用于传输的消息，消息使用框架后放入缓冲区中进行传输，由于传输的初始

就交换了各自的协议定义，因此即使传输双方使用的协议不同所传输的数据也能够正确解析。

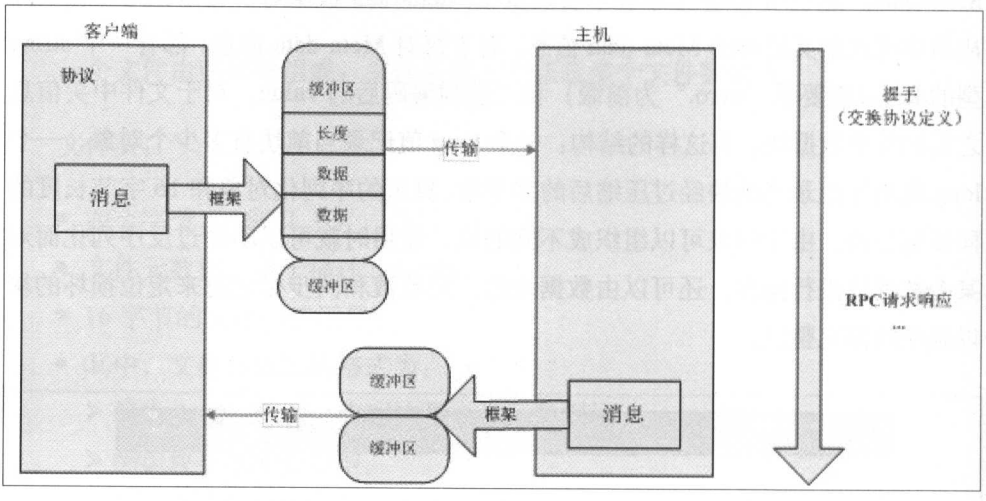


图 8.4 AvroRPC 过程

Avro 作为 RPC 框架来使用，客户端希望同服务器端交互时，就需要交换双方通信的协议，它类似于模式，需要双方来定义，在 Avro 中被称为消息(Message)。通信双方都必须保持这种协议，以便解析从对方发送过来的数据，这也就是传说中的握手阶段。

握手的过程是确保 Server 和 Client 获得对方的 Schema 定义，从而使 Server 能够正确反序列化请求信息，Client 能够正确反序列化响应信息。一般地，Server/Client 会缓存最近使用到的一些协议格式，所以，大多数情况下，握手过程不需要交换整个 Schema 文本。

消息从客户端发送到服务器端需要经过传输层，它发送消息并接收服务器端的响应。到达传输层的数据就是二进制数据。通常以 HTTP 作为传输模型，数据以POST方式发送到对方去。在Avro中，它的消息被封装成为一组缓冲区(Buffer)，类似于图 8.5 的模型。

每个缓冲区以四个字节开头，中间是多个字节的缓冲数据，最后以一个空缓冲区结尾。这种机制的好处在于，发送端在发送数据时可以很方便地组装不同数

据源的数据，接收方也可以将数据存入不同的存储区。还有，当往缓冲区中写数据时，大对象可以独占一个缓冲区，而不是与其他小对象混合存放，便于接收方方便地读取大对象。对象容器文件是 Avro 的数据存储的具体实现，数据交换则由 RPC 服务提供，与对象容器文件类似，数据交换也完全依赖 Schema，所以与 Hadoop 目前的 RPC 不同，Avro 在数据交换之前需要通过握手过程先交换 Schema。所有的 RPC 请求和响应处理都建立在已经完成握手的基础上。对于无状态的连接，所有的请求响应之前都附有一次握手过程；对于有状态的连接，一次握手完成，整个连接的生命期内都有效。

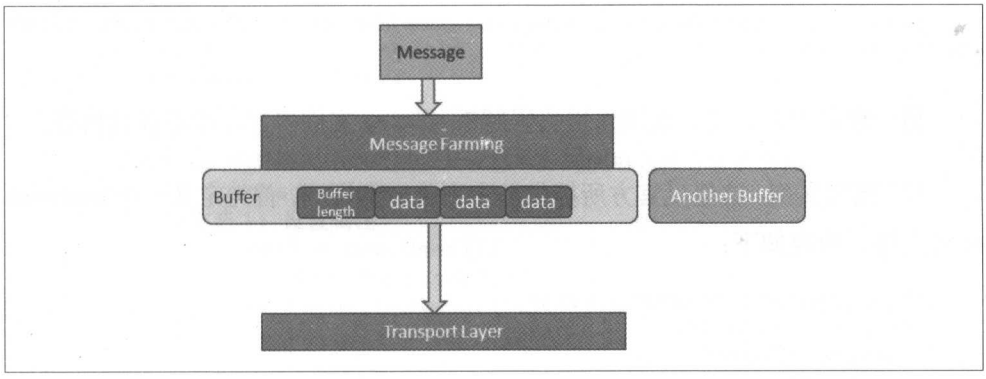


图 8.5 Avro 中消息传输流程

4．Avro使用

（1）Maven 项目构建 pom 示例，所需 jar 报如下：

```
<!-- 序列化需要的 jar -->
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.8.0</version>
</dependency>
<!-- rpc 通信需要的 jar -->
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-ipc</artifactId>
    <version>1.8.0</version>
</dependency>
```

Avro 不仅可以用做序列化和反序列化工具，更可以作为专业 PRC 服务框架。

对这两种功能，我们都可以有两种方式来实现：

一种是静态方式，采用 schema 文件或 protocol 文件使用 avro-tools-1.8.0.jar 来生成所需要的类，然后直接调用类里面的实现，生成静态类代码命令格式如下：

- 序列化生成静态类库命令格式如下：

```
java -jar avro-tools-1.8.0.jar compile schema <schema file>
<destination>
```

- RPC 生成静态类库命令格式如下：

```
java -jar avro-tools-1.8.0.jar compile protocol <schema file>
<destination>
```

另一种是动态方式，直接采用代码解析 schema 文件内容，动态设置内容。

(2) 模式实例：这里以官方所提供的序列化实例说明，首先定义一个 user.avsc 模式文件，内容如下：

```
{"namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
 ]
}
```

生成静态类过程如下：

```
bogon:tmp a$ java -jar avro-tools-1.8.0.jar compile schema
user.avsc .
Input files to compile:
  user.avsc
log4j:WARN No appenders could be found for logger
(AvroVelocityLogChute).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#
noconfig for more info.
```

上述命令执行完，即可看到当前目录下所生成的目录 example/avro 下有个 User.java。

(3)代码实例:这里给读者朋友演示分别使用静态方式和动态方式序列化代码。

```
package org.light.avro;

import java.io.File;
import java.io.IOException;
import org.apache.avro.Schema;
import org.apache.avro.file.DataFileReader;
import org.apache.avro.file.DataFileWriter;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericDatumReader;
import org.apache.avro.generic.GenericDatumWriter;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.DatumWriter;
import org.apache.avro.specific.SpecificDatumReader;
import org.apache.avro.specific.SpecificDatumWriter;
import example.avro.User;

public class TestAvro {
    //使用生成的静态类序列化数据到硬盘，分别使用 3 种方式实例化
    public void addUserCompile(){
        //方法 1：设置属性
        User user1 = new User();
        user1.setName("王 light");
        user1.setFavoriteNumber(66);
        user1.setFavoriteColor("浅蓝色");

        //方法 2：构造器
        User user2 = new User("魏 Sunny", 88, "red");

        // 方法 3：通过 builder 构造实例化
        User user3 = User.newBuilder()
                    .setName("王 Sam")
                    .setFavoriteColor("blue")
                    .setFavoriteNumber(2011)
                    .build();

        DatumWriter<User>userDatumWriter =
                    new SpecificDatumWriter<User>(User.class);
        DataFileWriter<User>dataFileWriter =
                    new DataFileWriter<User>(userDatumWriter);
        try {
            dataFileWriter.create(user1.getSchema(),
                    new File("/Users/a/Desktop/tmp/ users.avro"));
            dataFileWriter.append(user1);
            dataFileWriter.append(user2);
            dataFileWriter.append(user3);
            dataFileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
```

```
    }
}

public void deserUserCompile(){
    // 从硬盘文件反序列化 User
    DatumReader<User>userDatumReader =
                    new SpecificDatumReader<User> (User.class);
    DataFileReader<User>dataFileReader = null;
    User user = null;
    try {
        dataFileReader = new DataFileReader<User>(
                new File("/Users/a/Desktop/tmp/ users.avro"),
                        userDatumReader);
        while (dataFileReader.hasNext()) {
            // Reuse user object by passing it to next().
            //This saves us from allocating and garbage
            // collecting many objects for files with
            // many items.
            user = dataFileReader.next(user);
            System.out.println(user);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

//动态加载 schema 文件序列化 User 到硬盘
public void addUserDynamic(){
    Schema schema = null;
    try {
        schema = new Schema.Parser().parse(
                new File("/Users/a/Desktop/tmp/ user.avsc"));
    } catch (IOException e) {
        e.printStackTrace();
    }
    GenericRecord user1 = new GenericData.Record(schema);
    user1.put("name", "王成光");
    user1.put("favorite_number", 2012);
    user1.put("favorite_color", "blue");

    GenericRecord user2 = new GenericData.Record(schema);
    user2.put("name", "王占平");
    user2.put("favorite_number", 1998);
    user2.put("favorite_color", "green");

    // Serialize user1 and user2 to disk
    File file = new File("/Users/a/Desktop/tmp/userDyn.avro");
    DatumWriter<GenericRecord>datumWriter =
            new GenericDatumWriter<GenericRecord>(schema);
    DataFileWriter<GenericRecord>dataFileWriter =
             new DataFileWriter<GenericRecord>(datumWriter);
    try {
```

```
            dataFileWriter.create(schema, file);
            dataFileWriter.append(user1);
            dataFileWriter.append(user2);
            dataFileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    //动态加载 schema 从硬盘文件反序列化 User
    public void deserUserDynamic(){
        Schema schema = null;
        try {
                schema = new Schema.Parser().parse(
                    new File("/Users/a/Desktop/tmp/user. avsc"));
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize users from disk
        DatumReader<GenericRecord>datumReader =
                  new GenericDatumReader<GenericRecord>(schema);
        File file = new File("/Users/a/Desktop/tmp/userDyn.avro");
        DataFileReader<GenericRecord>dataFileReader = null;
        GenericRecord user = null;
        try {
            dataFileReader = new DataFileReader<GenericRecord>
(file, datumReader);
            while (dataFileReader.hasNext()) {
                // Reuse user object by passing it to next().
                // This saves us from allocating and garbage
                // collecting many objects for files with
                // many items.
                user = dataFileReader.next(user);
                System.out.println(user);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void test(){
        this.addUserCompile();
        this.deserUserCompile();
        System.out.println("=========以上是静态类方式序列化,以下是动态
加载方式序列化==========");
        this.addUserDynamic();
        this.deserUserDynamic();
    }

    public static void main(String[] args) {
        TestAvro ta = new TestAvro();
        ta.test();
```

```
        }
    }
```

输出如下：

```
    {"name": "王 light", "favorite_number": 66, "favorite_color": "浅
蓝色"}
    {"name": "魏 Sunny", "favorite_number": 88, "favorite_color": "red"}
    {"name": "王 Sam", "favorite_number": 2011, "favorite_color":
"blue"}
    =========以上是静态类方式序列化，以下是动态加载方式序列化==========
    {"name": "王成光", "favorite_number": 2012, "favorite_color":
"blue"}
    {"name": "王占平", "favorite_number": 1998, "favorite_color":
"green"}
```

当 Avro 用做 RPC 时，服务端支持的网络通信协议有：NettyServer、HttpServer，具体使用可以参考 https://github.com/phunt/avro-rpc-quickstart。

8.2.3 Dubbo/Dubbox 介绍

1. 基本认识

Dubbo 是淘宝开源的一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程服务调用方案，以及 SOA 服务治理方案。本质上是个服务调用的框架，说白了就是个远程服务调用的分布式框架。其核心部分包含：

- 远程通信：提供对多种基于长连接的 NIO 框架抽象封装，包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式。
- 集群容错：提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。
- 自动发现：基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

利用 Dubbo，可以帮您完成：

- 透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单

配置，没有任何 API 侵入。

- 软负载均衡及容错机制，可在内网替代 F5 等硬件负载均衡器，降低成本，减少单点。
- 服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的 IP 地址，并且能够平滑添加或删除服务提供者。

Dubbox（即 Dubbo eXtensions）是当当网根据自身的需求，为 Dubbo 实现的一些新的扩展功能。其主要新功能包括：

- 支持 REST 风格远程调用（HTTP + JSON/XML）：基于非常成熟的 JBossRestEasy 框架，在 Dubbo 中实现了 REST 风格（HTTP + JSON/XML）的远程调用，以显著简化企业内部的跨语言交互，同时显著简化企业对外的 Open API、无线 API 甚至 AJAX 服务端等的开发。事实上，这个 REST 调用也使得 Dubbo 可以对当今特别流行的“微服务”架构提供基础性支持。另外，REST 调用也达到了比较高的性能，在基准测试下，HTTP + JSON 与 Dubbo 2.x 默认的 RPC 协议（即 TCP + Hessian2 二进制序列化）之间只有 1.5 倍左右的差距。
- 支持基于 Kryo 和 FST 的 Java 高效序列化实现：基于当今比较知名的 Kryo 和 FST 高性能序列化库，为 Dubbo 默认的 RPC 协议添加新的序列化实现，并优化调整了其序列化体系，比较显著地提高了 Dubbo RPC 的性能。
- 支持基于嵌入式 Tomcat 的 HTTP remoting 体系：基于嵌入式 tomcat 实现 Dubbo 的 HTTP remoting 体系（即 dubbo-remoting-http），用以逐步取代 Dubbo 中旧版本的嵌入式 Jetty，可以显著提高 REST 等的远程调用性能，并将 Servlet API 的支持从 2.5 升级到 3.1。（注：除了 REST，Dubbo 中的 Web Services、Hessian、HTTP Invoker 等协议都基于这个 HTTP remoting 体系。）
- 升级 Spring：将 Dubbo 中 Spring 由 2.x 升级到目前最常用的 3.x 版本，减少项目中版本冲突带来的麻烦。
- 升级 ZooKeeper 客户端：将 Dubbo 中的 zookeeper 客户端升级到最新的版本，以修正老版本中包含的 bug。

注：Dubbox 和 Dubbo 2.x 是兼容的，没有改变 Dubbo 的任何已有的功能和配置方式（除了升级了 Spring 之类的版本）。

2. 技术要领

（1）整体架构

结合 Dubbo 官方文档（http://dubbo.io/User+Guide-zh.htm），Dubbo 框架设计一共划分了 10 个层，而最上面的 Service 层是留给实际想要使用 Dubbo 开发分布式服务的开发者实现业务逻辑的接口层[5]。图 8.6 中左边淡蓝背景的为服务消费方使用的接口，右边淡绿色背景的为服务提供方使用的接口，位于中轴线上的为双方都用到的接口。

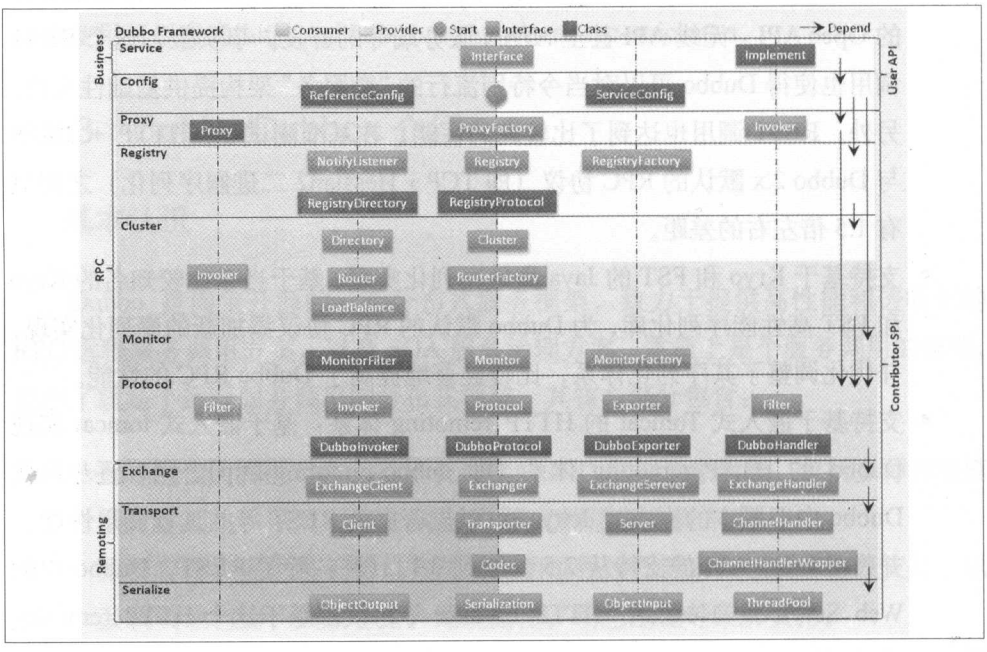


图 8.6 Dubbo 逻辑架构

我们分别理解一下框架分层架构中，各个层次的设计要点：

5 Dubbo 架构设计详解 http://shiyanjun.cn/archives/325.html

- 服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。
- 配置层（Config）：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为中心，可以直接 new 配置类，也可以通过 spring 解析配置生成配置类。
- 服务代理层（Proxy）：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton，以 ServiceProxy 为中心，扩展接口为 ProxyFactory。
- 服务注册层（Registry）：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 RegistryFactory、Registry 和 RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。
- 集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心，扩展接口为 Cluster、Directory、Router 和 LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。
- 监控层（Monitor）：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFactory、Monitor 和 MonitorService。
- 远程调用层（Protocol）：封将 RPC 调用，以 Invocation 和 Result 为中心，扩展接口为 Protocol、Invoker 和 Exporter。Protocol 是服务域，它是 Invoker 暴露和引用的主功能入口，它负责 Invoker 的生命周期管理。Invoker 是实体域，它是 Dubbo 的核心模型，其他模型都向它靠拢，或转换成它，它代表一个可执行体，可向它发起 invoke 调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。
- 信息交换层（Exchange）：封装请求响应模式，同步转异步，以 Request 和 Response 为中心，扩展接口为 Exchanger、ExchangeChannel、ExchangeClient 和 ExchangeServer。
- 网络传输层（Transport）：抽象 mina 和 Netty 为统一接口，以 Message 为中心，扩展接口为 Channel、Transporter、Client、Server 和 Codec。
- 数据序列化层（Serialize）：可复用的一些工具，扩展接口为 Serialization、ObjectInput、ObjectOutput 和 ThreadPool。

Dubbo 对于服务提供方和服务消费方，从框架的 10 层中分别提供了各自需要

关心和扩展的接口，构建整个服务生态系统（服务提供方和服务消费方本身以一个以服务为中心），其流程架构如图 8.7 所示。

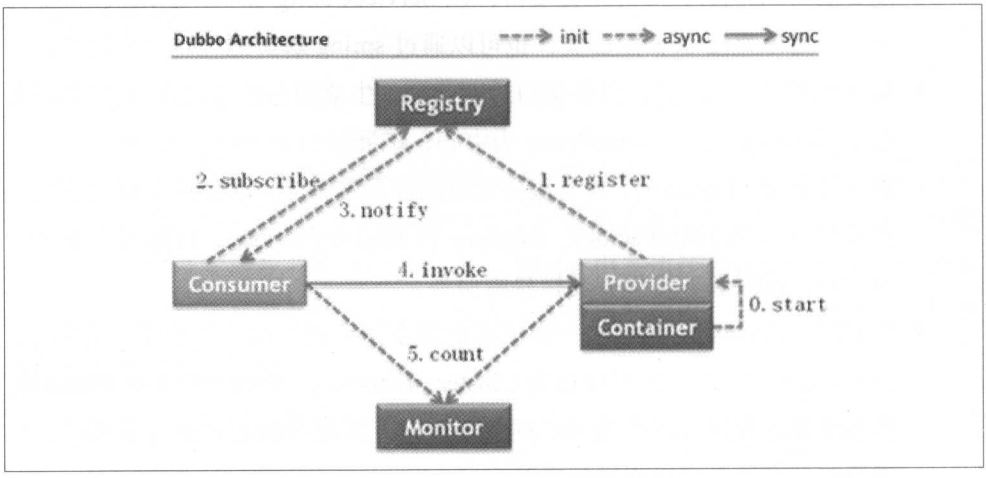


图 8.7 Dubbo 流程架构

节点角色说明：

- **Provider**：暴露服务的服务提供方。
- **Consumer**：调用远程服务的服务消费方。
- **Registry**：服务注册与发现的注册中心。
- **Monitor**：统计服务的调用次调和调用时间的监控中心。
- **Container**：服务运行容器。

调用关系说明：

- 0. 服务容器负责启动，加载，运行服务提供者。
- 1. 服务提供者在启动时，向注册中心注册自己提供的服务。
- 2. 服务消费者在启动时，向注册中心订阅自己所需的服务。
- 3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
- 4. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

- 5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

（2）使用方式

Dubbo 采用全 Spring 配置方式，透明化接入应用，对应用没有任何 API 侵入，只需用 Spring 加载 Dubbo 的配置即可，Dubbo 基于 Spring 的 Schema 扩展进行加载。如果不想使用 Spring 配置，而希望通过 API 的方式进行调用（不推荐）。

目前 Dubbo 及 Dubbox 有一定的使用广度，Dubbo 虽然是一套比较完善的 SOA 解决方案，但由于自 2013 年起，淘宝官方就没有再更新，且目前 Dubbo 仅支持 Java，由于 JDK8、JDK9 的推进，为了今后项目维护和可持续发展，虽然 Dubbo 值得借鉴学习，但笔者不建议大家使用 Dubbo 用在今后的项目中。

8.2.4 GRPC/ProtoBuf 介绍

1. 基本认识

GRPC 是一个高性能、开源和通用的 RPC 框架，其由 Google 主要面向移动应用开发并基于 HTTP/2 协议标准而设计，基于 ProtoBuf（Protocol Buffers）序列化协议开发，且支持众多开发语言。

（1）GRPC 具有的重要特征有：

- 强大的 IDL 特性

gRPC 使用 ProtoBuf 来定义服务，ProtoBuf 是由 Google 开发的一种数据序列化协议（类似于 XML、JSON、hessian）。ProtoBuf 能够将数据进行序列化，并广泛应用在数据存储、通信协议等方面。不过，当前 gRPC 仅支持 Protobuf，且不支持在浏览器中使用。由于 gRPC 的设计能够支持支持多种数据格式，所以读者能够很容易实现对其他数据格式（如 XML、JSON 等）的支持。

尽管 protocol buffers 对于开源用户来说已经存在了一段时间，虽然你可以使

用 proto2（当前默认的 protocol buffers 版本），官方建议在 gRPC 里使用 proto3，因为这样你可以使用 gRPC 支持全部范围的语言，并且能避免 proto2 客户端与 proto3 服务端交互时出现的兼容性问题，反之亦然。

- 支持多种语言 gRPC 支持多种语言，并能够基于语言自动生成客户端和服务端功能库。目前，在 GitHub 上已提供了 C 版本 grpc、Java 版本 grpc-java 和 Go 版本 grpc-go，其他语言的版本正在积极开发中，其中 grpc 支持 C、C++、Node.js、Python、Ruby、Objective-C、PHP 和 C#等语言，grpc-java 已经支持 Android 开发。
- 基于 HTTP/2 标准设计。

由于 gRPC 基于 HTTP/2 标准设计，所以相对于其他 RPC 框架，gRPC 带来了更多强大功能，如双向流、头部压缩、多复用请求等。这些功能给移动设备带来重大益处，如节省带宽、降低 TCP 链接次数、节省 CPU 使用和延长电池寿命等。同时，gRPC 还能够提高了云端服务和 Web 应用的性能。gRPC 既能够在客户端应用，也能够在服务器端应用，从而以透明的方式实现客户端和服务器端的通信和简化通信系统的构建。

（2）ProtoBuf 的 proto3 和 proto2 区别

ProtoBuf 目前最新版是 protobuf-3.0.0-beta-3，总体说，proto3 比 proto2 支持更多语言但更简洁，去掉了一些复杂的语法和特性，更强调约定而弱化语法。如果是首次使用 Protobuf，建议使用 proto3。下面列出 proto3 与 proto2 的重大调整[6]。

- 首行版本标记：proto3 的编译器同时支持 proto2 语法和 proto3 的语法，如果你的 proto 文件没有添加 syntax 说明的话，用这个版本的编译器会报错，提示你默认 proto2 支持，如果要确认用 proto3，请在第一行非空白非注释行添加语法标记：syntax = "proto3";

- 字段规则调整：proto2 中，只保留 repeated 标记数组类型，optional 和 required

6 Google Protobuf3 版本介绍 http://www.tuicool.com/articles/YNni6rv

都被去掉了。proto3 字段类型默认只有两种：singular 和 repeated。

- map 支持：map 编写格式为：map<key_type, value_type> map_field = N；
- 字段 default 标记不能使用了：在 proto2 中，可以使用 default 选项为某一字段指定默认值。在 proto3 中，字段的默认值只能根据字段类型由系统决定。也就是说，默认值全部是约定好的，而不再提供指定默认值的语法。
- 枚举默认值一定是 0：proto2 里的默认值是枚举的第一个 value 对应的值，不一定为 0；proto3 在你定义 value 时，强制要求第一个值必须为 0。这个修改为避免隐患还是有帮助的。
- 移除了对分组的支持：分组的功能完全可以用消息嵌套的方式来实现，并且更清晰。在 proto2 中已经把分组语法标注为“过期”了。这次也算清理垃圾了。
- 泛型描述支持：any 类型，可以代表任何类型，可以先读进来，再进行解析。
- 增加了 JSON 映射特性，当前 JSON 的流行有其充分的理由，语言的活力来自于与时俱进。
- 增加了多种语言支持：js,objc,ruby,C#等。

2. 技术要领

GRPC 官方文档：http://www.grpc.io/，其对中文文档：http://doc.oschina.net/grpc。

（1）GRPC 客户端和服务端通信逻辑

在 GRPC 里客户端应用可以像调用本地对象一样直接调用另一台不同的机器上服务端应用的方法，使得您能够更容易地创建分布式应用和服务。与许多 RPC 系统类似，GRPC 也是基于以下理念：定义一个服务，指定其能够被远程调用的方法（包含参数和返回类型）。在服务端实现这个接口，并运行一个 GRPC 服务器来处理客户端调用。在客户端拥有一个*存根*能够像服务端一样的方法。

如图 8.8 所示，gRPC 客户端和服务端可以在多种环境中运行和交互。从 google 内部的服务器到你自己的笔记本，并且可以用任何 GRPC 支持的语言来编写。所以，你可以很容易地用 Java 创建一个 GRPC 服务端，用 Go、Python、Ruby 来创建客户端。此外，Google 最新 API 将有 GRPC 版本的接口，使你很容易地将 Google

的功能集成到你的应用里。

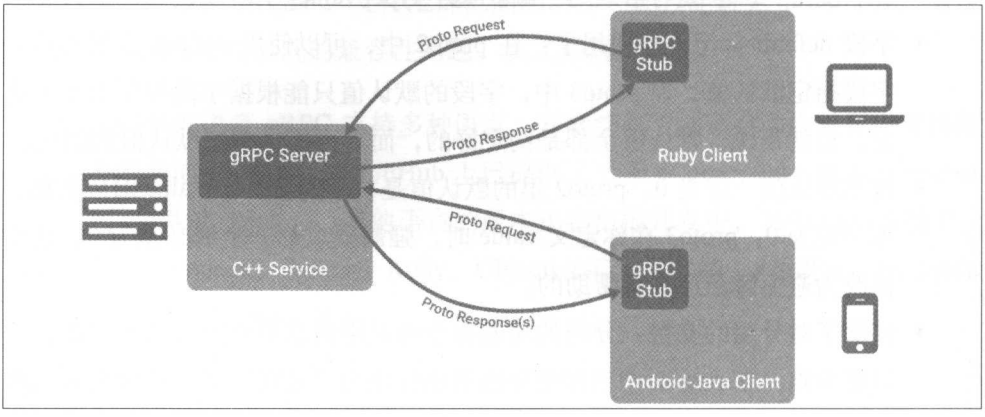


图 8.8 GRPC 通信逻辑流程示意图

(2) 服务定义

正如其他 RPC 系统，gRPC 基于如下思想：定义一个服务，指定其可以被远程调用的方法及其参数和返回类型。gRPC 默认使用 protocol buffers 作为接口定义语言，来描述服务接口和有效载荷消息结构。如果有需要的话，可以使用其他替代方案。以下是官方所提供的一个 helloworld.proto 样例。

```
syntax = "proto3";
option java_package = "io.grpc.examples";

service HelloService {
    rpcSayHello (HelloRequest) returns (HelloResponse);
}
message HelloRequest {
  required string greeting = 1;
}
message HelloResponse {
  required string reply = 1;
}
```

如果在.proto 文件中没有显示的 java_package 参数，那么就会使用缺省的 proto 包（通过"package" 关键字指定）。但是，因为 proto 包一般不是以域名翻转的格式命名，所以它不是好的 Java 包。如果我们用其他语言通过.proto 文件生成代码，java_package 是不起任何作用的。

利用 protobuf 生成客户端代码格式如下（假定生成）：

```
protoc --proto_path=proto 文件目录 --java_out=生成代码目录
helloworld.proto
```

这里不同语言对应的输出标示如下：

```
C++ (include C++ runtime and protoc)<--->cpp_out
Java    <--->java_out
Python<--->python_out
Objective-C <--->objc_out
C#  <--->csharp_out
JavaNano    <--->javanano_out
JavaScript<--->    js_out
Ruby    <--->ruby_out
Go  <--->go_out
PHP <--->php_out
```

(3) 服务类型及工作流程

要定义一个服务，你必须在你的.proto 文件中指定 service，然后在我们的服务中定义 rpc 方法，指定它们的请求的和响应类型。GRPC 允许你定义 4 种类型的 service 方法：

- **单项 RPC**：即客户端发送一个请求给服务端，从服务端获取一个应答，就像一次普通的函数调用。例如：

```
rpcSayHello(HelloRequest) returns (HelloResponse){}
```

其工作流程如下：

- 一旦客户端通过桩调用一个方法，服务端会得到相关通知，通知包括客户端的元数据、方法名、允许的响应期限（如果可以的话）。
- 服务端既可以在任何响应之前直接发送回初始的元数据，也可以等待客户端的请求信息，到底哪个先发生，取决于具体的应用。
- 一旦服务端获得客户端的请求信息，就会做所需的任何工作来创建或组装对应的响应。如果成功的话，这个响应会和包含状态码以及可选的状态信息等状态明细及可选的追踪信息返回给客户端。

➢ 假如状态是 OK 的话，客户端会得到应答，这将结束客户端的调用。

- **服务端流式 RPC**：即客户端发送一个请求给服务端，可获取一个数据流用来读取一系列消息。客户端从返回的数据流里一直读取，直到没有更多消息为止。从例子中可以看出，通过在响应类型前插入 stream 关键字，可以指定一个服务器端的流方法。

  例如：

```
rpcLotsOfReplies(HelloRequest) returns (stream HelloResponse){}
```

  其工作流程：服务端流式 RPC 除了在得到客户端请求信息后返回一个应答流之外，与单项 RPC 一样。在发送完所有应答后，服务端的状态详情（状态码和可选的状态信息）和可选的跟踪元数据被发送回客户端，以此来完成服务端的工作。客户端在接收到所有服务端的应答后也完成了工作。

- **客户端流式 RPC**：即客户端用提供的一个数据流写入并发送一系列消息给服务端。一旦客户端完成消息写入，就等待服务端读取这些消息并返回应答。通过在*请求*类型前指定 stream 关键字来指定一个客户端的流方法。例如：

```
rpcLotsOfGreetings(stream HelloRequest) returns (HelloResponse) {}
```

  其工作流程：客户端流式 RPC 也基本与单项 RPC 一样，区别在于客户端通过发送一个请求流给服务端，取代了原先发送的单个请求。服务端通常（但并不必须）会在接收到客户端所有的请求后发送回一个应答，其中附带有它的状态详情和可选的跟踪数据。

- **双向流式 RPC**：即两边都可以分别通过一个读写数据流来发送一系列消息。这两个数据流操作是相互独立的，所以客户端和服务端能按其希望的任意顺序读写，例如：服务端可以在写应答前等待所有的客户端消息，或者它可以先读一个消息再写一个消息，或者是读写相结合的其他方式。每个数据流里消息的顺序会被保持。通过在请求和响应前加 stream 关键字去制定方法的类型。例如：

```
rpcBidiHello(stream HelloRequest) returns (stream HelloResponse){}
```

  其工作流程：双向流式 RPC，调用由客户端调用方法来初始化，而

服务端则接收到客户端的元数据、方法名和截止时间。服务端可以选择发送回它的初始元数据或等待客户端发送请求。下一步怎样发展取决于应用，因为客户端和服务端能在任意顺序上读写。这些流的操作是完全独立的。例如服务端可以一直等直到它接收到所有客户端的消息才写应答，或者服务端和客户端可以像“乒乓球”一样：服务端后得到一个请求就回送一个应答，接着客户端根据应答来发送另一个请求，依此类推。

3. Java客户端使用

Maven 项目构建 pom 示例，所需 jar 报如下：

```
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-all</artifactId>
    <version>0.14.0</version>
</dependency>
```

服务端和客户端实例，可以参考官方实例代码：

```
https://github.com/grpc/grpc-java/tree/master/examples/src/main/
java/io/grpc/examples/helloworld
```

其中有 hellworld 的 client 和 server 以及支持 JSON 序列化的完整实例代码，并有相关说明。

GPRC 于 2015 年 2 月推出，目前还在快速发展期，虽然功能比较诱人，且出身名门，但实际生产环境使用时，尚需谨慎，毕竟任何新技术都需要监测和逐渐完善。

8.2.5 ZeroC ICE

1. 基本认识

ZeroC ICE 是指 ZeroC 公司的 ICE（Internet Communications Engine）中间件平台，支持广泛的语言，包括 C++, C#, Java, JavaScript, Objective-C, PHP, Python, and Ruby 等，可以部署到 Linux, OS X, Windows, Android, iOS 等，对于客户端和

服务端程序的开发提供了很大的便利。目前 ICE 平台中包括 ICE，ICE-E，ICE Touch，目前最新版 3.6.2，其中：

ICE 为主流平台设计，包括 Windows 和 Linux，支持广泛的语言，包括 C++，Java，C#（和其他.NET 的语言，例如 Visual Basic），Python，Ruby，PHP 和 ActionScript。也包括所有的 ICE 服务，例如 ICE Grid，ICEStorm 等。

ICE-E 是 ICE 在资源受限的平台上的一个实现，支持 C++和嵌入式操作系统，例如 Windows CE，Linux。ICE-E 本身不包含任何服务，但是可以利用在 ICE 上提供的各种服务。因此，通过 ICE-E，移动设备也能无缝地集成到分布式系统中。

ICE Touch 是为 iPhone 和 iPod Touch 开发的版本，包括 Object-C 映射，支持 iPhone OS，并为 MAC OS X 开发图形界面应用程序提供完整的 Cocoa 框架访问。

Zero-ICE 相对比较成熟，目前阿里开源的关系型数据库分布式处理系统 MyCAT 底层就是依靠 ICE 通信的。

2．技术要领

（1）核心组成：以下为 Zero ICE 的几个组成部分及功能介绍[7]。

Slice ICE 的规范语言，跟 CORBA 的 IDL（Interface Definition Language）等价的东西，是一种中间语言。Slice 建立了客户端和服务器端共同遵守的契约：接口。Slice 也用来描述对象持久数据。

Slice Compilers 的规范语言可以映射成多种编程语言。目前 ICE 支持 C++，Java，Python，PHP，C#和 VB 的语言映射。ICE 的客户端和服务器端协同工作，而不会知道分别实现的是何种编程语言。

Servant：**ice.Object** 的实例化对象叫做 servant，rpc 调用的就是 servant 对象，是服务端必须实现的模块，因此 servant 需要线程安全。

---

7 Zero ICE 的概念、组成与服务 http://www.cnblogs.com/chuncn/archive/2013/04/23/3037191.html

ICE 的核心库：在众多的特性当中，ICE 核心库通过一个高效的协议（包含 TCP/UDP 层上协议压缩）来管理所有的通信任务，为多线程服务器提供了一个灵活的线程池，并且有特别的功能来支持上百万对象的可扩展性。

IceUtil 一些常用的功能函数集。例如 Unicode 处理和多线程编程，是用 C++ 语言写成的。

ICEBox 是一个专用于 ICE 应用的应用服务器，它可以协调多个应用组件启动和停止。ICEBox 可以方便地运行和管理动态加载、共享库或 Java 类的形式 ICE 的服务。应用组件可以用动态链接库的形式发布而不是一个进程，这就减轻了系统的负载。例如，你可以在一个 JVM 中运行若干个应用组件而不是有多个进程，每一个进程都有自己的 JVM。

ICEPack 是一个成熟的服务激活和部署工具。ICEPack 能大大简化在异构网络之间部署应用的复杂性。只要简单地编写 XML 格式的一个部署描述文件，ICEPack 就能自动处理剩下的工作。ICEPack 是 Ice 的定位服务。当使用间接绑定时，用来将符号化的适配器名称转换为协议—地址对。除了定位服务之外，ICEPack 还提供了如下的服务：

- ICEPack 允许你注册一个自动启动的服务：即当客户端进行请求时，服务器不需要处于运行状态，只要第一个客户端进行请求时，服务会自动启动。
- ICEPack 支持脚本描述部署，可以轻易地配置包含了若干个服务的复杂的应用。
- ICEPack 提供了简单的对象查找服务，允许客户端获取他们感兴趣的对象代理。

Freeze 提供了 ICE Servants 对象的自动持久性。通过几行代码，一个应用就可以生成一个高度可扩展的逐出器（evictor）来高效地管理持久对象。

FreezeScript 在大的软件项目里，持久对象的数据类型改变很常见。为了最小化这些变化的影响，FreezeScript 提供了相应的工具来检查和移植 Freeze 生成的数据库。这些工具支持 XML 格式的配置脚本，易于使用。

ICESSL 用于 ICE 核心的动态的 SSL 传输插件，提供了认证、加密和消息完整性，使用工业标准的 SSL 协议来实现。

Glacier 面向对象中间件平台的一个最大的挑战是安全性和防火墙。Glacier 是 ICE 的防火墙解决方案，它允许客户和服务器通过防火墙安全的通信。客户-服务器的通信通过使用公钥认证完全加密，并且通信是双向的。Glacier 提供了相互认证和安全的会话管理支持。Glacier 认证和过滤客户的请求并允许服务器通过安全的方式回调客户端对象。结合 ICESSL 的使用，Glacier 提供了强大的安全解决方案，既安全，又易于配置管理。

ICEStorm 是一个支持联盟的发布-订阅消息服务，它减除了客户端和服务器的耦合度。本质上说，ICEStorm 作为一个事件分发的交换机运行。发布者将事件发给服务，ICEStorm 按照顺序将事件传递给订阅者。使用这种方法，一个事件发布者就可以把一个事件发布给多个订阅者。事件按照主题分类，订阅者可以指定他们感兴趣的主题。只有订阅者感兴趣的主题才会发送给订阅者。服务允许指定服务的质量，从而允许应用在可伸缩性和性能之间进行适当的折中。如果你需要将信息发布到大量的应用组件，那么 IceStorm 是一个不错的选择。ICEStorm 减除了信息的发布者和订阅者之间的耦合，同时也能重新发布已经发布的信息。另外，ICEStorm 可以作为联合服务运行，即多个服务的实例可以运行在不同的机器上，从而降低了 CPU 的负载。

ICEPatch 是一个软件修补和分发的服务。ICEPatch 允许你轻松地把软件的更新发布给客户。客户连接到 ICEPatch 然后请求更新一个特定的应用。服务就自动检查客户软件的版本然后下载需要更新的组件。而这些下载的组件都是放在一个压缩包里的，从而减少了带宽的占用。ICEPatch 自动更新在某个目录层次下的文件。只有需要更新的文件会下放到客户端，为了快速的下载更新，ICEPatch 使用的是高效的压缩算法。软件补丁也可以通过结合 Glacier 服务发布，这样可以让只有经过授权的客户才能下载软件更新。

ICEGrid 用于支持分布式网络服务应用，一个 ICEGrid 域由一个注册表（Registry）和任何数目的节点（Node）构成。注册表和节点一起合作管理一些信息以及包含一些应用（Application）的服务进程。每个应用程序分配到特定节点

的服务器。这个注册表维护了这些信息，注册表中的信息记录被持久化到数据库中，而节点负责启动和监测其指定的服务器进程。对于一个典型的配置，一个节点运行在一台计算机（称之为 Ice 服务器主机）。注册表并不消耗很多处理器时间，所以它常常是和一个节点运行在同一台计算机上的，事实上，注册表和一个节点可以运行在同一进程中。如果要想容错机制达到理想的状态，注册表也支持复制（Replication）功能使用主从式的设计。

从客户端应用程序的角度来看，注册表的主要责任是解决作为 Ice 定位服务的间接代理问题。因此，这方面的作用是非常明显的：当客户端第一次尝试使用一种间接代理，客户端的 Ice run time 连接注册表，并且将代理的符号信息转化为端点，使用这个端点允许客户端建立一个连接。尽管注册还提供了一些其他的功能，不仅仅是一个简单的查询表，一个定位请求可能提示一个节点自动启动目标服务，或注册表可能会根据每台电脑的负荷统计选择适当的端点。

间接代理的好处是：位置服务可以提供很大的功能，而客户端不需要任何额外的特定操作，这点和直接代理不同，客户端并不需要更多的服务器的地址和端口信息。只是间接代理在客户的第一次使用代理时增加了一些延迟，不过，以后所有的相互作用直接发生在客户端和服务器之间，所以成本是微不足道的。此外，间接代理的方式允许将已经部署的服务器迁移到不同的计算机上，而不需要更新客户端所持有的代理。

（2）客户端和服务端通信流程：图 8.9 显示了使用 ICE 作为中间件平台[8]，客户端及服务端的应用都是由应用代码及 ICE 的库代码混合组成的。**客户应用**及**服务器应用**分别对应的是客户端与服务端。**代理**是根据 Slice 定义的 ice 文件实现，它提供了一个向下调用的接口，提供了数据的序列化与反序化。ICE 的核心部分，提供了客户端与服务端的网络连接等核心通信功能，以及其他的网络通信功能的实现及可能的问题的处理，让我们在编写应用代码的时候不必去关注这一块，而专注于应用功能的实现。

8 ICE 简单介绍及使用示例 http://blog.csdn.net/fenglibing/article/details/6372444

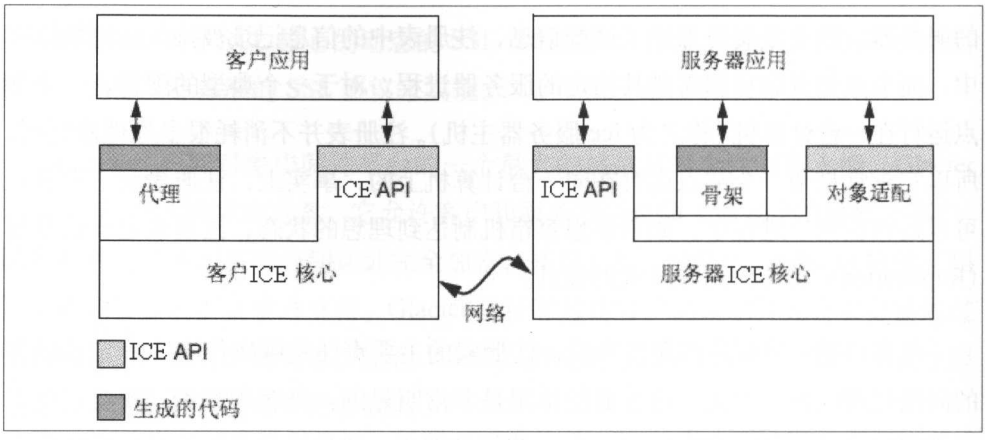


图 8.9 ICE client 和 Server 通信流程

ICE 采用的网络协议有 TCP、UDP 以及 SSL 三种，在调用模式上有好几种选择方案，并且每种方案正对不同的网络协议的特性做了相应的选择。

Oneway(单向调用)：客户端只需将调用注册到本地传输缓冲区（Local Transport Buffers）后就立即返回，不会等待调用结果的返回，不对调用结果负责。

Twoway（双向调用）：最通用的模式，同步方法调用模式，只能用 TCP 或 SSL 协议。

Datagram（数据报）：类似于 Oneway 调用，不同的是 Datagram 调用只能采用 UDP 协议而且只能调用无返回值和无输出参数的方法。

BatchOneway（批量单向调用）：先将调用存 在调用缓冲区里面，到达一定限额后自动批量发送所有请求（也可手动刷除缓冲区）。

BatchDatagram（批量数据报）：与上类似。

不同的调用模式其实对应着不动的业务，对于大部分的有返回值的或需要实时响应的方法，我们可能都采用 Twoway 方式调用，对于一些无需返回值或 者不依赖返回值的业务，我们可以用 Oneway 或者 BatchOneway 方式，例如消息通知；剩下的 Datagram 和 BatchDatagram 方式 一般用在无返回值且不做可靠性检查的业务上，例如日志。

3. java客户端实例

Maven 项目依赖包如下：

```
<dependency>
<groupId>com.zeroc</groupId>
<artifactId>ice</artifactId>
<version>3.6.2</version>
</dependency>
```

实例为一个参数为 string 返回值也为 string 的函数实例，其 ICE 描述为 IceDemo.ice：

```
module myDemo{
    interface Demo{
        string say(string s);
    };
};
```

其中，myDemo 为包名，生成 java 相关 client 和 Server 端命令如下：

slice2java IceDemo.ice。

运行此命令后，则在当前目录生成相关代码。Servant 实例 DemoI.java 完整代码如下：

```
package org.light.ice;
import Ice.Current;
import myDemo._DemoDisp;
public class DemoI extends _DemoDisp {
    private static final long serialVersionUID =
8823361341775010885L;

    @Override
    public String say(String s, Current __current) {
        System.out.println("---收到客户端请求参数："+s);
        return "Demo Test : "+s;
    }
}
```

Server 端完整实例 DemoServer.java。

```
package org.light.ice;
public class DemoServer {

    public static void main(String[] args) {
```

```
        int status = 0;
        // Communicator 实例，是 ice run time 的主句柄
        Ice.Communicator ic = null;
        try {
            // 调用 Ice.Util.Initialize()初始化 Ice run time
            System.out.println("初始化 ice run time...");
            ic = Ice.Util.initialize(args); // args 参数可传可不传
            // 创建一个对象适配器，传入适配器名和 10000 端口接收来的请求
            System.out.println("创建对象适配器，监听端口 10000...");
            Ice.ObjectAdapter adapter = ic
                            .createObjectAdapterWithEndpoints(
                             "DemoAdapter", "default -p 10000");
            // 实例化一个 DemoI 对象，为 Demo 接口创建一个 servant
            System.out.println("为接口创建 servant...");
            Ice.Object object = new DemoI();
            // 调用适配器的 add,告诉它有一个新的 servant，传递的参数
            //是刚才的 servant,这里的“FirstIceDemo”是 Servant 的名字
            System.out.println("对象适配器加入 servant...");
            adapter.add(object,
                    Ice.Util.stringToIdentity ("FirstIceDemo"));
          //调用 activate()方法，激活适配器。服务器开始处理来自客户的请求。
            System.out.println("激活适配器，服务器等待处理请求...");
            adapter.activate();
            // 这个方法挂起发出调用的线程，直到服务器终止为止。
            // 或我们自己发出一个调用关闭。
            ic.waitForShutdown();
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        } finally {
            if (ic != null) {
                ic.destroy();
            }
        }
        System.exit(status);
    }
}
```

Client 端完整实例 DemoClient.java。

```
package org.light.ice;
import Ice.AsyncResult;
import myDemo.DemoPrx;
import myDemo.DemoPrxHelper;
public class DemoClient {

    public static void main(String[] args) {
```

```
        int status = 0;
        // Communicator 实例
        Ice.Communicator ic = null;
        try {
            // 调用 Ice.Util.Initialize()初始化 Ice run time
            ic = Ice.Util.initialize(args);
            // 根据 servant 的名称以及服务器 ip、端口获取远程服务代理
            Ice.ObjectPrx base = ic.stringToProxy(
                    "FirstIceDemo:tcp -h 127.0.0.1 -p 10000");
            // 将上面的代理向下转换成一个 Printer 接口的代理
            DemoPrxfirstDemo = DemoPrxHelper.checkedCast(base);
            // 如果转换成功
            if (firstDemo == null) {
                throw new Error("Invalid proxy");
            }
            // 调用这个代理，将字符串传给它
            //异步调用： AMI
            AsyncResult ar = firstDemo.begin_say(
                        "王光的第一个 ICE 测试--异步调用");
            System.out.println(firstDemo.end_say(ar));
            //同步调用
            System.out.println(firstDemo.say(
                            "王光的第一个 ICE 测试--同步"));
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        } finally {
            if (ic != null) {
                ic.destroy();
            }
        }
        System.exit(status);
    }
}
```

上述示例只是 Zero-ICE 作为 RPC 基本工具的一个功能点展现，还有很多其他功能，尚需要读者自行深入研究。

8.3 Web Service 介绍及实践

当前 Web Service 主要有两种方式：一是 SOAP 协议方式，在这种方式下需要 WSDL，UDDI 等；二是 REST 方式，这种方式根本不需要 WSDL，UDDI 等。

当前看 REST 方式更加流行，更有前途。因此本节也重点给大家介绍 Rest。当前 Web Service 实现技术有很多种方式，仅仅基于 Java 的就有：基于 JWS 方式、基于 Jetty 方式、基于 SpringMVC 方式、基于 Axis2、CXF、XFire，以及 Jersey 等，相信读者朋友都希望开发轻量级、简单易用，所谓大道至简，所以此处重点给大家介绍前三种实现方案。

8.3.1 SOAP 和 Rest

1. SOAP介绍

SOAP 简单对象访问协议（SOAP，全写为 Simple Object Access Protocol）是交换数据的一种协议规范，在计算机网络 Web 服务（Web Service）中使用，交换带结构信息。它本身不具备传输性，但是可以使用 HTTP 协议，Socket 作为载体进行传输。一个典型的 SOAP 结构[9]为：

- SOAP 消息必须用 XML 来编码
- SOAP 消息必须使用 SOAP Envelope 命名空间
- SOAP 消息必须使用 SOAP Encoding 命名空间
- SOAP 消息不能包含 DTD 引用
- SOAP 消息不能包含 XML 处理指令

提到 SOAP 不得不提的是另外两个概念：WSDL 和 UDDI。WSDL（Web 服务描述语言，Web Services Description Language）是为描述 Web 服务发布的 XML 格式。SOAP 就是使用 WSDL 来描述 Web 服务的。UDDI 是统一描述、发现和集成（Universal Description, Discovery, and Integration）的缩写。它是一个基于 XML 的跨平台的描述规范，可以使世界范围内的企业在互联网上发布自己所提供的服务。SOAP，WSDL，UDDI 这三者是 W3C 中定义 Web Service 的三个核心组件。

- **SOAP**：一个基于 XML 的可扩展消息信封格式，需同时绑定一个传输用协

9 白话 REST-识别真假 REST http://www.uml.org.cn/soa/201606283.asp

议。这个协议通常是 HTTP 或 HTTPS，但也可能是 SMTP 或 XMPP。

- **WSDL**：一个 XML 格式文档，用以描述服务端口访问方式和使用协议的细节。通常用来辅助生成服务器和客户端代码及配置信息。
- **UDDI**：一个用来发布和搜索 Web 服务的协议，应用程序可借由此协议在设计或运行时找到目标 Web 服务。

三者可以相互独立，也可以相互融合使用，理想的应用场景是：

- **Web 服务提供者**：通过 SOAP 描述交互数据，使用 WSDL 描述服务器端口访问方式，在 UDDI 注册自己的 Web 服务，以方便调用者查找。
- **Web 服务的消费者**：在 UDDI 上查找 Web 服务，调用 WSDL 获得服务接口的访问方式和接口细节，使用 SOAP 交互数据。

2. REST介绍

REST 是一种架构风格，其核心是面向资源，REST 专门针对网络应用设计和开发方式，以降低开发的复杂性，提高系统的可伸缩性。REST 提出设计概念和准则为：

- 网络上的所有事物都可以被抽象为资源（resource）。
- 每一个资源都有唯一的资源标识，对资源的操作不会改变这些标识。
- 所有的操作都是无状态的。
- 传输 XML、JavaScript Object Notation（JSON），或同时传输这两者。

REST 简化开发，其架构遵循 CRUD 原则，该原则告诉我们对于资源（包括网络资源）只需要四种行为：创建、获取、更新和删除，就可以完成相关的操作和处理。您可以通过统一资源标识符（Universal Resource Identifier，URI）来识别和定位资源，并且针对这些资源而执行的操作是通过 HTTP 规范定义的。其核心操作只有 GET,PUT,POST,DELETE。由于 REST 强制所有的操作都必须是 stateless 的，这就没有上下文的约束，如果做分布式，集群都不需要考虑上下文和会话保持的问题，极大地提高系统的可伸缩性。

RESTful：RESTful Web 服务（也称为 RESTful Web API）是一个使用 HTTP

并遵循 REST 原则的 Web 服务，用于 Web 服务和动态 Web 应用程序的多层架构，可以实现可重用性、简单性、可扩展性和组件可响应性的清晰分离。目前主流 Web 2.0 服务提供者（包括 Yahoo、Google 和 Facebook）都对 REST 的采用，放弃基于 SOAP 和 WSDL 的接口。

8.3.2 JWS（JDK 自身实现 Web Service）

JWS 是 JDK6 起开始具备的功能，依靠 JWS 自身就可以完成 Web Service 服务。发布一个 JWS 服务实例，并从客户端调用相关服务 API[10]步骤如下：

- 在类上添加@Web Service 注解（此注解是 JDK1.6 提供的，位于 javax.jws.Web Service 包中）后，类中所有的非静态方法都将对外公布，不支持静态方法，final 方法。如果希望某个方法（非 static，非 final）不对外公开，可以在方法上添加@WebMenthod(exclude=true)，防止对外公开。
- 通过 EndPoint（端点服务）发布一个 Web Service。

EndPoint 是 JDK 提供的一个专门用于发布服务的类，该类的 publish 方法接收两个参数：一个是本地的服务地址，二是提供服务的类。位于 javax.xml.ws.Endpoint 包中。

- 根据发布的服务地址，利用 JDK 自身命令 wsimport 生成客户端代码。这里说明下：参数 s 指定源代码生成目录，参数 d 指定 class 输出目录。
- 利用客户端代码，完成客户端和服务端通信，调用相关服务 API。

这里需要注意，服务类中只有 public method 才会被发布，且一个服务类至少应该有一个可供发布的 method，否则 JWS 就会提示你常见的典型错误，形如“com.sun.xml.internal.ws.model.RuntimeModelerException: 由类 org.light.ws.jdk.HelloJws 定义的 Web 服务不包含任何有效的 WebMethods”。

以下为 JWS 使用完整示例代码，其中 HelloJws.java 为定义的服务 API 及服

10 Java 项目中发布 WebService 服务 http://blog.csdn.net/hanxuemin12345/avticle/details/40163757

务发布类，JwsClient.java 为客户端实现类。

HelloJws.java 完整代码如下，它包含 3 个方法，作为示例，特意给读者展示防止对外公开@WebMenthod(exclude=true)使用。

```
package org.light.ws.jdk;

import javax.jws.WebMethod;
import javax.jws.Web Service;
import javax.xml.ws.Endpoint;

//@Web Service 是一个注解，用在类上指定将此类发布成一个 ws.
@Web Service
public class HelloJws {

    public String sayHello(String name){
        return "Hi, "+name+", 你好! ";
    }

    public String getBornDesc(String name, String homeTown){
        return name+" 的故乡是 "+homeTown;
    }

    //添加 exclude=true 后，HelloWord2()方法不会被发布
    @WebMethod(exclude=true)
    public String getResumeDesc(String name, String degree){
        return name+" 的学历是 "+degree;
    }

    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/hellojws",
                         new HelloJws());
    }
}
```

以上是 JWS 服务发布完整实例，运行该类后，可以直接在浏览器地址端输入其发布的服务描述 http://localhost:8080/hellojws?wsdl 查看。

图 8.10 为 JDK 命令 wsimport 利用刚发布的 WSDL 生成客户端代码流程示意图。这里只是一个示例，真实应用时，一般都会利用域名，DNS 会自动映射到后端的几个服务节点，自动实现负载均衡，提高服务的高可用性。

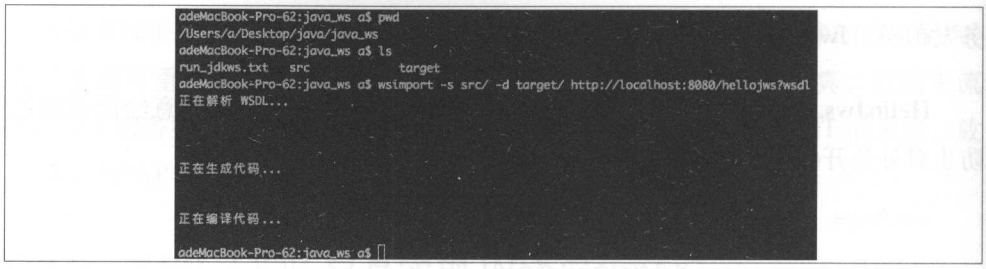


图 8.10 JWS 生成客户端代码

```
package org.light.ws.jdk.client;

import org.light.ws.jdk.HelloJws;
import org.light.ws.jdk.HelloJwsService;

public class JwsClient {

        public void run(){
            //获取服务端连接，此处“HelloJws”为客户端生成的接口
            HelloJwshjws =
                        new HelloJwsService().getHelloJwsPort();
            System.out.println(hjws.sayHello("Sam Wang"));
            System.out.println(hjws.getBornDesc("Sam Wang",
                                    "生在北京，老家山东省郓城县"));
    }

    public static void main(String[] args) {
        JwsClient jc = new JwsClient();
        jc.run();
    }
}
```

JWS 一般会把服务端和客户端视为 2 个独立工程，因为服务端发布的服务定义类 HelloJws.java，和 JWS 根据发布的 WSDL 所生成的接口 HelloJws.java 同名，且默认同包，当然可以在产生客户端代码时，显示指定生成的包结构。JWS 的好处在于不依赖第三方包，使用简单，方便调试开发。

8.3.3 Jetty：嵌入式 Servlet 容器

谈到当前开源 Servlet 容器，大家首先想到的可能是 Tomcat，但是，Tomcat 并不孤单，我们还有 Jetty。Jetty 是一个开源的 Servlet 容器，它为基于 Java 的 web 内容，例如 JSP 和 Servlet 提供运行环境。Jetty 是使用 Java 语言编写的，它的 API 以一组 JAR 包的形式发布。开发人员可以将 Jetty 容器实例化成一个对象，可以

迅速为一些独立运行（stand-alone）的 Java 应用提供网络和 Web 连接。

Jetty 作为可选的 Servlet 容器只是一个额外的功能，而它真正出名，是因为它是作为一个可以嵌入到其他的 Java 代码中的 Servlet 容器而设计的。这就是说，开发小组将 Jetty 作为一组 Jar 文件提供出来，因此你可以在你自己的代码中将 Servlet 容器实例化成一个对象并且可以操纵这个容器对象。这也是本书作者所推崇的，简单易学易用。

下面 JettyWS.java 为使用 Jetty 嵌入式 Servlet 容器完成的一个 Web Service，如图 8.11 所示，你可以在运行 JettyWS 后，直接在浏览器地址栏输入预先设定的参数，得到相关数据。

当前 Jetty 最新版 9.39，为方便大家使用，一般在 pom.xml 增加 Maven 依赖：

```
<dependency>
    <groupId>org.eclipse.jetty.aggregate</groupId>
    <artifactId>jetty-all</artifactId>
    <version>9.3.9.v20160517</version>
    <type>pom</type>
</dependency>
```

便可以一次引进 Jetty 所有相关开发所需 Jar，下面为 JettyWS.java 源代码：

```
package org.light.jetty;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.servlet.ServletHandler;

public class JettyWS {

    public static void main(String[] args) throws Exception {
        // 实例化 1 个监听端口 8080 的 Jetty 服务对象
        Server server = new Server(8080);
        // ServletHandler 继承自 ScopedHandler，是 Jetty 中用于存储
        // 所有 Filter、FilterMapping、Servlet、ServletMapping
        //的地方，以及用于实现一次请求所对应的 Filter 链和 Servlet 执行
        //流程的类。对 Servlet 的框架实现中，它也被认为是 Handler 链的末
        //端，因而在它的 doHandle()方法中没有调用 nextHandle()方法。
        ServletHandler handler = new ServletHandler();
```

```
        server.setHandler(handler);
        // 为特定 Servlet 映射到特定访问路径
        handler.addServletWithMapping(HelloServlet.class,
                                                    "/hello");
        //服务启动
        server.start();
        // 当前线程和主线程绑定，和进程共进退
        server.join();
    }

    @SuppressWarnings("serial")
    public static class HelloServlet extends HttpServlet {
        @Override
        protected void doGet(HttpServletRequest request,
                HttpServletResponse response)
                throws ServletException, IOException {
            response.setContentType("text/html");
            response.setStatus(HttpServletResponse.SC_OK);
            String name = request.getParameter("name");
            String age = request.getParameter("age");
            PrintWriterpwt = response.getWriter();
            pwt.write("姓名：<h1>"+name+"</h1>年龄："+age);
            pwt.flush();
            pwt.close();
        }
    }
}
```

图 8.11 Jetty 嵌入式 Web Service 效果图

从上图可以看出，有了 Jetyy 嵌入式容器，以后可以很方便地启动 Web Service，也便于调试程序，以及更加直观、便捷的验证。当然，生产环境下一般配合域名使用或者借助 Nginx 等完成服务的集群负载均衡。

8.3.4 基于 Spring MVC

Spring MVC 是 Spring 为展现层提供的基于 MVC 设计理念的优秀的 Web 框架，是目前最主流的 MVC 框架之一。Spring3.0 后全面超越 Struts2，成为最优秀的 MVC 框架，它通过一套 MVC 注解，让 POJO 成为处理请求控制器，而无须实

现任何接口，采用了松散耦合的可插拔组件结构，比其他 MVC 框架更具扩展性和灵活性，且支持 REST 风格 URL 请求。SpringMVC 目前最新版是 4.2.6，目前用得最多的还是 3.2.x。

Spring 4.X 对 Spring MVC 部分做的增强是最多的，提供了一些视图解析器的 MVC 标签实现简化配置、提供了 GroovyWebApplicationContext 用于 Groovy Web 集成，提供了 Gson、protobuf 的 HttpMessageConverter，提供了对 groovy-templates 模板的支持、JSONP 的支持、对 Jackson 的@JsonView 的支持等。

Spring 涉及知识非常多，除了最基本的 SpringMVC、SpringFramework，还有其衍生的很多附属组件，像 Spring-Data、Spring-AMQP、Spring-ES、Spring－Solr、Spring-Hadoop、Spring-Shell 等，Spring 大有一统天下之势。Spring 也专门提供了 Spring－WS 子项目，专门针对 Web service，其主要基于 SOAP，类似于前述 JWS。我们重点读者介绍基于 SpringMVC 的 Web Service 核心实现的几个注解组件[11]：

@Controller 注解标识一个控制器，一般一个 Controller 类处理一种资源，所以每个 Controller 类都会加@RequestMapping 注解。

@RequestMapping 这是最重要的一个注解，用于为控制器指定可以处理那些 URL 请求，处理 HTTP 请求地址映射，可用于类或方法上。

- 类定义处：提供初步的请求映射信息，相对于 Web 应用根目录；
- 方法处：提供进一步的细分映射信息，相对于类定义处 URL，若类定义处未标注，则方法标处标记的 URL 相对于 Web 应用根目录。

其常用属性如下所示。

- value：指定请求的地址。
- method：指定请求的 method 类型，GET、POST、PUT、DELETE 等。
- params：指定 request 中必须包含某些参数值时，才让该方法处理。

11 详解 SpringMVC4 常用的那些注解 http://aijuans.iteye.com/blog/2160141

@PathVariable 映射 URL 绑定的占位符（带占位符的 URL 是 Spring3.0 开始具备的功能，更有助于 REST），通过它可以将占位符参数绑定到控制器处理方法入参中。这里需要说明另外一个注解@RequestParam，也是将 request 中参数值绑定到 control 方法参数里，二者的不同之处在于:

- @PathVariable 的 url 是这样的：http://host:port/.../path/参数值。
- @RequestParam 的 url 是这样的：http://host:port/.../path?参数名=参数值。

@RequestBody 用于读取 Request 请求的 body 数据，使用 HttpMessageConverter 将数据转换为 Java 对象，再把对象绑定到 controller 中方法的参数上。

@ResponseBody 用于将 Controller 中方法返回的对象，使用 HttpMessageConverter 转换为指定格式数据，再写入到 Response 对象的 body 数据区。

下面给出作者所列出的文章管理的实例，限于篇幅，这里只列出其中核心 Controller 代码实现完整源代码：ArticleController.java。

```
package org.light.web.controller;

import java.io.UnsupportedEncodingException;
import java.util.Date;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import org.light.web.model.Article;
import org.light.web.model.ArticlePage;
import org.light.web.service.ArticleService;

/**
 * @author light wang
 * @param<T>
 */
@Controller
```

```
@RequestMapping("/article")
public class ArticleController {

    /**
     * 文章管理服务
     */
    @Autowired
    private ArticleServicearticleService;

    /**
     * 跳转到添加文章
     */
    @RequestMapping("addShow")
    public String addShow() {
        return "articleAdd";
    }

    /**
     * 添加文章
     * @param title 标题
     * @param content 内容
     */
    @RequestMapping("add")
    public String add(String title, String content) {
        Article article = new Article();
        article.setTitle(title);
        article.setContent(content);
        article.setCreatetime(new Date());
        articleService.add(article);
        return "redirect:list.do";
    }

    /**
     * 显示一篇文章
     * @param id 文章 id
     * @返回到文章详情页面
     */
    @RequestMapping("/show")
    public String show(int id, HttpServletRequest request) {
        Article article = articleService.getById(id);
        request.setAttribute("article", article);
        return "articleShow";
    }

    /**
     * 显示一篇文章 json 格式
     * @param id 文章 id
     * @return 文章 json 数据
     */
    @RequestMapping("/show/json")
    @ResponseBody
```

```
public Article show(int id) {
    Article article = articleService.getById(id);
    return article;
}

/**
 * 转向文章列表页面
 */
@RequestMapping("/list")
public String list(HttpServletRequest request) {
    List<Article> articles = articleService.getAll();
    request.setAttribute("articles", articles);
    return "articleList";
}

/**
 * 返回文章列表内容，对象内容转为 json 格式
 */
@RequestMapping("/list/json")
@ResponseBody
public List<Article> list() {
    List<Article> articles = articleService.getAll();
    return articles;
}

/**
 * 直接返回 List<Map<String, String>>格式的文章列表内容
 */
@RequestMapping("/list/map")
@ResponseBody
public List<Map<String, String>>listArtMap() {
    List<Map<String, String>> articles
                = articleService. getDataMaplist();
    System.out.println(articles);
    return articles;
}

/**
 * 删除文章
 * @param id
 * @return
 */
@RequestMapping("/delete/{id}")
public String delete(@PathVariable("id") int id) {
    articleService.delete(id);
    return "redirect:list.do";
}

@RequestMapping("jsonfeed")
@ResponseBody
public Object getJSON(Model model) {
    model.addAttribute("items", articleService.getAll());
```

```
        model.addAttribute("status", 0);
        return model;
    }

    @RequestMapping("toAjax")
    @ResponseBody
    public String toAjax(String txt) {
        try {
            txt = new String(txt.getBytes("ISO-8859-1"),"UTF-8");
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return "toAjax "+txt;
    }

    @RequestMapping("health")
    @ResponseBody
    public int healthCheck(int code) {
        return code;
    }
}
```

该示例给读者朋友展示 SpringMVC 的 Web Service 中几个注解使用，分别列出了重定向、返回内容和从路径获取参数等用法。这里只是给读者一个入门示例，抛砖引玉，希望读者能深入学习 Spring 相关知识。

8.3.5 其他 Web Service 框架

除了上述几个框架，还有下面的几个框架[12]需要大家了解的。

Axis2 是 Apache 下的一个重量级 Web Service 框架，准确说它是一个 Web Service/SOAP/WSDL 的引擎，是 Web Service 框架的集大成者，它不但能制作和发布 Web Service，而且可以生成 Java 和其他语言版 Web Service 客户端和服务端代码。这是它的优势所在。但是，这也不可避免地导致了 Axis2 的复杂性，使用过的开发者都知道，它所依赖的包数量和大小都是很惊人的，打包部署发布都比较麻烦，不能很好地与现有应用整合为一体。但是如果你要开发 Java 之外别的语言客户端，Axis2 所提供的丰富工具将是你不二的选择。

12 有了 Java 6，还需要 Axis2、XFire、CXF 吗？http://lavasoft.blog.51cto.com/62575/228552/

XFire 是一个高性能的 Web Service 框架，在 Java 6 之前，它的知名度甚至超过了 Apache 的 Axis2，XFire 的优点是开发方便，与现有的 Web 整合很好，可以融为一体，并且开发也很方便。但是对 Java 之外的语言，没有提供相关的代码工具。XFire 后来被 Apache 收购了，原因是它太优秀了，收购后，随着 Java 6 JWS 的兴起，开源的 Web Service 引擎已经不再被看好，渐渐地都败落了。

CXF 是 Apache 旗下一个重磅的 SOA 简易框架，它实现了 ESB（企业服务总线）。CXF 来自于 XFire 项目，经过改造后形成的，就像目前的 Struts2 来自 WebWork 一样。可以看出 XFire 的命运会和 WebWork 的命运一样，最终会淡出人们的视线。CXF 不但是一个优秀的 Web Service/SOAP/WSDL 引擎，也是一个不错的 ESB 总线，为 SOA 的实施提供了一种选择方案，当然它也不是最好的，仅仅实现了 SOA 架构的一部分。

8.4 总结

本章主要给读者介绍当前微服务架构中互联网行业主流通信框架，无论是其中 RPC 服务框架：Thrift/nifty、Avro、Dubbo、GRPC 和 Zero-ice，还是 Web Service 服务框架：jws、Jetty 嵌入式 servlet 或是基于 SpringMVC，均为当前主流开发技术，当然仅仅这么一章篇幅不可能面面俱到，也仅仅是一个引子，所以这里通过理论和开发实例，引领读者快速入门，后期要想在生产环境下用好，还需要读者自己深入学习。

本章中涉及各个服务框架实例代码均已放在 github（https://github.com/lrtdc/book_ldrtc）。

9 第9章 综合实例：新闻推荐中的用户画像近实时更新

前面章节陆续给读者介绍了当前大数据环境下，分布式实时计算各环节中常用的各个组件技术，本章将会以新闻个性化推荐中用户画像实时更新为例，向读者进一步详细介绍如何将前述的各个相关组件整合在一起，从而完成一个具体的业务需求。本章作者以自己多年实践经验向读者介绍当前主流个性化推荐系统架构的组成部分，然后向读者详细介绍实例设计及核心开发。

9.1 个性化推荐系统组成

个性化推荐系统是一门由数据挖掘和机器学习综合的学科，它必须能够基于用户之前的口味和喜好提供其感兴趣的精准推荐，而且这种口味和喜欢的收集必须尽量少地需要用户的显示参与工作。本节向读者介绍个性化推荐系统的各个组成部分。

图 9.1 所示为当前主流个性化推荐系统的整体架构及其相关逻辑组成部分。从整个图可以看出，整个推荐系统将会是一个闭环，从而使得推荐系统结合相关算法可以完成自我学习，下面将会给大家向读者详细介绍每个部分及其实现方式。

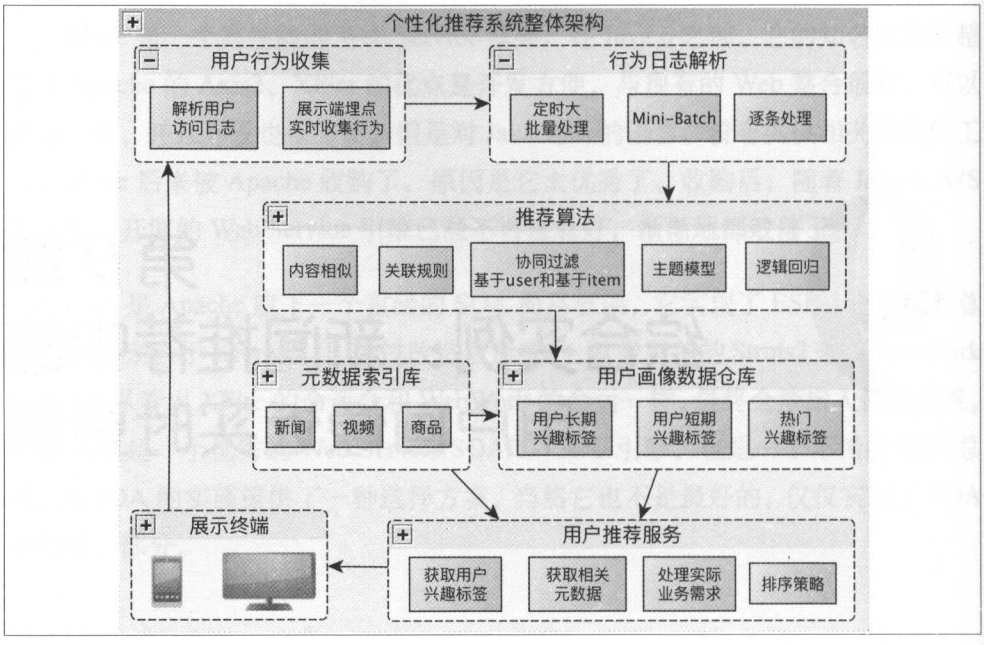


图 9.1 个性化推荐系统整体架构

9.1.1 用户行为收集

如图 9.1 所示，当前主流个性化推荐系统收集用户行为的方式一般来自两种方式，即网站用户访问日志和展示终端埋点，下面分别加以说明。

1. 用户访问日志

当前一般网站访问日志都会记录在 Nginx 日志中，在 Nginx 中有自己默认的日志格式，如下内容：

```
log_format access '$remote_addr - $remote_user [$time_local] "$request" ''$status $body_bytes_sent "$http_referer" ''"$http_user_agent" $http_x_forwarded_for';
```

其中每项说明如下：

- $remote_addr：客户端的 IP 地址（如果中间有代理服务器，那么这里显示的 IP 就为代理服务器的 IP 地址），可以告诉我们访问用户来自哪里，可以

统计海量用户的地区分布；

- $remote_user：用于记录远程客户端的用户名称（一般为“-”）；
- $time_local：用于记录访问时间和时区，可以帮我们准确统计出每个时间段的用户访问频次；
- $request：用于记录请求的 URL，以及请求方法，这个字段非常重要，目前一般网站访问 URL 都是 Rest 格式，它可以直接帮我们分析出用户的行为类型及资源类型，比如如果是电商商品，可以分析出用户的行为属于浏览／收藏／分享／加入购物车等，还可以进一步分析出商品 SKU，进而知道用户对其所属品牌和品类等感兴趣；如果是搜索类请求 URL，那么就会包含很多过滤条件，更能分析出用户的当前兴趣点；
- $status：响应状态码，标识请求响应状态；
- $body_bytes_sent：给客户端发送的文件主体内容大小；
- $http_refer：可以记录用户是从哪个链接跳转过来，统计用户来源，站外跳转过来，站内跳转或者是伪造用户的网络爬虫等；
- $http_user_agent：用户所使用的代理（一般为操作系统、浏览器等），可以分析出使用每个类型设备的用户使用情况；
- $http_x_forwarded_for：可以记录客户端 IP，通过代理服务器来记录客户端的 IP 地址。

网站访问日志，一般不会有遗漏，记载信息量非常大，是一个非常宝贵的五星财富，是比较基础的数据，因此也是我们数据挖掘的一个重要来源。

2．展示端的埋点

有些重要的用户行为，不适合在网站访问日志中呈现，所以这类用户行为一般都是通过在网站前端埋点收集的，比如用户的分享、收藏及转发行为，一般采用 Ajax 异步请求，不会在网站日志看到。

这里所谓的埋点，是指采用事先准备完善的 Web Service 接口，在相应各个终端初始化加载相应服务，一旦某个行为触发，则立马 Ajax 异步提交该行为给 Web Service API，为了提高效率，服务端一般采用异步处理，使用 MQ 统一收集数据，

服务端同时启动几个服务进程实时监听 MQ 数据，从而完成相应处理。当下 MQ 用得比较多的当属 Kafka 和 RabbitMQ，前述章节已经给读者朋友做了专门介绍。

9.1.2 行为日志解析

如上节所述，用户行为来源主要是用户的网站访问日志和展示终端埋点异步收集的日志，目前看，行为日志解析主要有 3 种方式：定时大批量处理、Mini-Batch（小批量处理）和逐条处理。

1．定时大批量处理

这种方式主要针对用户的网站访问日志，目前主流网站访问日志均为 Nginx 日志，如上节所述基本格式，一般根据企业实际业务需要，每隔若干小时或每天，甚至每周集中处理一次，一般可以采用 Python 或 Java，比较适合采用 Hadoop 离线计算。这种方式一般配合实时计算，对于本章主题所说的用户画像，一般用来周期性地修正用户画像标签权重因实时计算带来的偏差。

2．逐条处理

这种方式主要针对展示终端埋点所实时收集的用户行为数据处理，一般借助 MQ 异步处理；当然也可以针对网站访问日志采用类似于 Linux 基本命令 tailf（实时读取文件末尾新增数据行），也同样可以实现逐条处理，相比较这种方式有一定延迟。逐条处理方式适用于对实时性要求非常高的场合，但相应地，尤其是在高并发期间，会大大增加服务端的计算压力。这种方式虽然有一定适用范围，比如当前热门主流实时计算框架 Storm，默认就是对收到的每条数据即时处理，但在生产环境下，更多的是使用接下来给大家重点介绍的 Mini-Batch，像作者本人所参与的实时计算项目中，一般都是在服务器端 buffer 固定条数或固定几秒的数据，每当达到临界点便即刻进行小批量处理。

3．Mini-Batch（小批量处理）

这个单词是借鉴自大名鼎鼎的 SparkStreaming 处理思想，也是作者本人所提

倡的处理方式，作者本人所独立研发的轻量级分布式实时计算框架 light_drtc 设计思想就是 Mini-Batch。这里的小批量处理，是相对于前面的定时大批量而言的，一般 Mini-Batch 只会间隔分钟级、秒级甚至微秒级处理一次。Mini-Batch 处理的技巧在于，将在间隔很小的时间段内的用户行为作为一个独立的数据集，统一处理。这样给服务端提交的计算任务就会远远小于逐条处理时的任务数量。这种处理方式对于前述两种用户行为收集方式都适用，也是目前行业广为实用的方式。

为了让读者朋友更加清晰地理解此处的用户行为解析，作者以之前项目中的一个 Nginx 网站访问日志解析为示例向大家介绍行为日志解析的方法。此示例使用 Python 完成，首先根据指定目录下文件数量，启动同等数量的子进程，每个子进程处理一个文件，文件内容为原始 Nginx 日志，解析好的内容存入 Mongo，下面是示例核心代码 **access_log_parser.py**。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#引入解析日志所需第三方依赖
import re
import httplib
import sys
import httpagentparser as hap
import gzip
from datetime import datetime
#python 多进程处理模块
import multiprocessing
#进程池中使用队列
from multiprocessing.managers import BaseManager
#导入多进程模块中的进程池和进程
from multiprocessing import Pool,Process
#本实例相应的 mongodb 操作 dao
from search_mongo import SearchMongo
#引入网站资源 url 前缀定义
from yg_urls import *

#设置系统编码格式
default_encoding = 'utf-8'
if sys.getdefaultencoding() != default_encoding:
    reload(sys)
    sys.setdefaultencoding(default_encoding)
#网站域名测试域名
MAIN_URL = 'http://www.light.org'
#排除掉不需要解析的资源，提高解析效率
EXCLUDE_URL = ['/images/','/js/','/css/']
```

```
    #访问日志解析正则范式，类似perl5的分组匹配模式
    ACCESS_LOG_PATTERN    =    re.compile(r'(?P<ip>[0-9\.]+)    -    -
\[(?P<visit_time>.*)\] \"(?P<current_url>.*)\" (?P<status_code>[0-9]+)
(?P<size>[0-9]+)      \"(?P<src_url>.*)\"      \"(?P<session_id>.*)\"
\"(?P<user_id>.*)\" \"(?P<user_agent>.*)\"')

    #日志源文件目录
    SRC_DIR='/data/applogs/access_logs/'
    #日志解析后将源文件移到备份目录
    DEST_DIR='/data/applogs/access_logs_bp/'

    #实例化Mongo访问Dao
    SMG = SearchMongo()
    #获取日志解析存取的表名
    ACCESS_TABLE = SMG.getMongoTable()

    #python 对象操作
    class AccessLogParser(object):
        def __init__(self):
            pass

        #逐行解析每条日志
        def parserLine(self,logLine):
            logDict = None
            m = ACCESS_LOG_PATTERN.match(logLine, 0)
            if m:
                #获取分组匹配的字典
                log_map = m.groupdict()
                if (log_map.has_key('session_id')
                    and log_map ['session_id']!='-') \
                    and (log_map.has_key('current_url') \
                    and log_map ['current_url']!='-'):
                    for up in EXCLUDE_URL:
                        if log_map['current_url'].find(up)>0:
                            return None
                    cur_url      =      self.__parser_cur_url(log_map
['current_url'])
                    if cur_url==None or cur_url.find(MAIN_URL)<0:
                        return None
                    url_parser = get_place(cur_url)
                    if url_parser:
                        logDict = {}
                        logDict['sid']    =    self.__parse_sessionid
(log_map['session_id'])
                        logDict['curl'] = url_parser
                        if log_map.has_key('src_url') \
                            and log_map['src_url']!='-':
                            src_url = log_map['src_url']
                            if src_url.find(MAIN_URL)>0:
                                surl_parser    =    get_place(log_map
['src_url'])
                                if surl_parser:
```

```
                                logDict['surl'] = surl_parser
                        if      log_map.has_key('user_id')      and
log_map['user_id']!='-':
                            logDict['uid']   =   self.__parser_userid
(log_map['user_id'])
                        if log_map.has_key('visit_time'):
                            vtms   =   self.__parse_visit_time(log_map
['visit_time'])
                            if vtms:
                                logDict['vtime']=self.__get_vdtime
(vtms)
                                logDict['vweek']=self.__get_vweek
(vtms)
                        if  log_map.has_key('ip')  and  log_map['ip']!
='-':
                            logDict['ip']=log_map['ip']
                        if log_map.has_key('status_code'):
                            logDict['status']=int(log_map
['status_code'])
                        if log_map.has_key('user_agent'):
                            brw_os=self.__parse_user_agent(log_map
['user_agent'])
                            if brw_os and len(brw_os)>0:
                                logDict['bwos'] = brw_os
                    log_map=None
            return logDict

        #返回访问来源网址
        def __parser_refer(self,refer_url):
            return refer_url

        #解析当前访问 URL
        def __parser_cur_url(self,cur_url):
            vt_url = cur_url.split(' ')
            if vt_url and len(vt_url)==3:
                fwurl = vt_url[1]
                return fwurl
            else:
                return None

        #获取会话 ID
        def __parse_sessionid(self, ssid):
            first_index = ssid.find('.')
            ssid = ssid[first_index+1:]
            first_index = ssid.find('.')
            ssid = ssid[:first_index]
            return ssid

        #获取注册用户 ID
        def __parser_userid(self, uid):
            first_index = uid.find('.')
            uid = uid[first_index+1:]
```

```
            if len(uid)==32:
                return uid
            else:
                return None

        #获取访问时间，并格式化
        def __parse_visit_time(self, vtime):
            result = None
            if vtime:
                vtms = vtime.split(' ')
                result  =  datetime.strptime(vtms[0],  '%d/%b/%Y:%H:
%M:%S')
            return result

        #根据访问时间获取年周
        def __get_vweek(self,vtms):
            return int(vtms.strftime('%y%W'))

        def __get_vdtime(self,vtms):
            return int(vtms.strftime('%y%m%d%H%M'))

        #解析 user_agent，获得访问用户所用设备的 OS 和浏览器
        def __parse_user_agent(self,usg):
            browser_os = None
            try:
                if usg:
                    browser_os = hap.detect(usg)
            except Exception, inst:
                print inst, 'Error user-agent is: \t',usg
                return None
            else:
                return browser_os

        #解析指定的单个日志文件
        def run_one(self, logFile):
            begin = datetime.now()
            hostIdIndex=logFile.find('_')
            hwebId = int(logFile[3:hostIdIndex])
            hIndex = logFile.find("-")+1
            logDate = logFile[hIndex:]
            logDate = logDate.replace('_','')
            logDate=logDate[2:]
            #打开二进制日志文件
            #glog = gzip.open(logFile,'rb')
            #打开普通文件
            glog = open(SRC_DIR+logFile,'r')
            lineNum = 0
            #循环，逐行读取并解析
            while True:
                line = glog.readline()
                lineNum += 1
                if lineNum%100000==0:
```

```
                print logFile, '-->', lineNum
            if not line:
                break
            lineDict = self.parserLine(line)
            if lineDict:
                lineDict['jxTime']=int(logDate)
                lineDict['serverId']=hwebId
                try:
                    #解析结果入库
                    ACCESS_TABLE.insert(lineDict)
                except Exception, inst:
                    print 'Error in insert into mongo : ',lineDict
        glog.close()
        shutil.move(SRC_DIR+logFile, DEST_DIR+logFile)
        end = datetime.now()
        print logFile,' 解析完毕,共耗时: ',(end-begin).seconds

    #从指定目录获取文件个数，启动同等数量子进程，并行解析每个日志文件入口
    def run(self):
        begin = datetime.now()
        #logDir = '/data/applogs/access_logs/'
        logList = os.listdir(SRC_DIR)
        mps = []
        for lf in logList:
            mps.append(multiprocessing.Process(target=
self.run_one, args=(lf,)))
        for ps in mps:
            ps.start()
        for ps in mps:
            ps.join()
        end = datetime.now()
            print '本批访问日志解析完毕,共耗时:',(end-begin).seconds

if __name__=="__main__":
    alp = AccessLogParser()
    alp.run()
```

上述程序为日志解析核心源代码，也顺便给大家演示 Python 在文件处理方面的优势。同样功能，如果用 Java，代码量至少是其 2 倍。

9.1.3 常用推荐算法

当前行业主流推荐算法如图 9.1 所示的几种，在一个比较完善的推荐系统里，一般都是若干种推荐算法的综合使用，根据 A／B 测试，最终以合适的比例综合推荐结果给用户展示。

1．内容相似

（1）适用场景

基于内容的推荐方法就是根据用户过去的浏览记录来向用户推荐用户没有接触过的推荐项，它是推荐算法中最基础、也是非常重要的一个推荐产品特性，基于内容的推荐算法思路比较简单，它的原理大概分为 3 步[1]：

- 为每个物品（Item）构建一个物品的属性资料（Item Profile）。
- 为每个用户（User）构建一个用户的喜好资料（User Profile）。
- 计算用户喜好资料与物品属性资料的相似度，相似度高意味着用户可能喜欢这个物品，相似度低往往意味着用户不喜欢这个物品。

选择一个想要推荐的用户“U”，针对用户 U 遍历一遍物品集合，计算出每个物品与用户 U 的相似度，选出相似度最高的 k 个物品，将它们推荐给用户 U。基于内容相似的推荐使用场景也非常广泛，下面列举几个典型场景：

- 电商网站商品详情页某个角落的相关商品；
- 门户网站新闻详情页某个角落的相关新闻；
- 视频等多媒体网站视频详情页某个角落的相关视频等。

（2）算法介绍

相似度度量（Similarity），即计算个体间的相似程度，相似度度量的值越小，说明个体间相似度越小，相似度的值越大说明个体差异越大。对于多个不同的文本或者短文本对话消息要来计算它们之间的相似度如何。一个好的做法就是将这些文本中词语，映射到向量空间，形成文本中文字和向量数据的映射关系，通过计算几个或者多个不同向量的差异的大小，来计算文本的相似度。下面介绍几个常用相似度计算方法。

- **向量空间余弦相似度**（Cosine Similarity）

1 一个简单的基于内容的推荐算法 http://www.cnblogs.com/exlsunshine/p/4214357.html

向量空间余弦相似度是用向量空间中两个向量夹角的余弦值作为衡量两个个体间差异的大小的度量[2]。向量，是多维空间中有方向的线段，如果两个向量的方向一致，即夹角接近零，那么这两个向量就相近。而要确定两个向量方向是否一致，这就要用到余弦定理计算向量的夹角。

余弦定理描述了三角形中任何一个夹角和三个边的关系。给定三角形的三条边，可以使用余弦定理求出三角形各个角的角度。假定三角形的三条边为 a，b 和 c，对应的三个角为 A，B 和 C，如果将三角形的两边 b 和 c 看成是两个向量，那么角 A 的余弦为：

$$\cos A = \frac{<b,c>}{|b\|c|}, \quad sim(X,Y) = cos\theta = \frac{\vec{x}\cdot\vec{y}}{\|x\|\cdot\|y\|}$$

其中分母表示两个向量 b 和 c 的长度，分子表示两个向量的内积。

举个具体的例子，假如新闻 X 和新闻 Y 对应向量分别是：$x1, x2, ..., x6400$ 和 $y1, y2, ..., y6400$，则它们之间的余弦距离可以用它们之间夹角的余弦值来表示：

$$\cos\theta = \frac{x_1y_1 + x_2y_2 + \cdots + x_{6400}y_{6400}}{\sqrt{x_1^2 + x_2^2 + \cdots + x_{6400}^2}\cdot\sqrt{y_1^2 + y_2^2 + \cdots + y_{6400}^2}}$$

当两条新闻向量夹角余弦等于 1 时，这两条新闻完全重复（用这个办法可以删除爬虫所收集网页中的重复网页）；当夹角的余弦值接近于 1 时，两条新闻相似（可以用做文本分类）；夹角的余弦越小，两条新闻越不相关。

- **修正余弦相似度**（Adjusted Cosine Similarity）

余弦相似度更多的是从方向上区分差异，而对绝对的数值不敏感，因此没法衡量每个维度上数值的差异，会导致这样一种情况：用户对内容评分，按 5 分制，X 和 Y 两个用户对两个内容的评分分别为（1,2）和（4,5），使用余弦相似度得到

2 余弦距离、欧氏距离和杰卡德相似性分析 http://www.cnblogs.com/chaosimple/archive/2013/06/28/3160839.html

的结果是 0.98，两者极为相似。但从评分上看 X 似乎不喜欢 2 这个内容，而 Y 则比较喜欢，余弦相似度对数值的不敏感导致了结果的误差，需要修正这种不合理性就出现了修正余弦相似度，即所有维度上的数值都减去一个均值，比如 X 和 Y 的评分均值都是 3，那么调整后为（-2,-1）和（1,2），再用余弦相似度计算，得到 -0.8，相似度为负值并且差异不小，但显然更加符合现实。修正余弦相似度公式[3]为 (Adjusted Cosine Similarity)：

$$sim(i,j)=\frac{\sum_{u\in U}(R_{u,i}-\bar{R}_u)(R_{u,j}-\bar{R}_u)}{\sqrt{\sum_{u\in U}(R_{u,i}-\bar{R}_u)^2}\sqrt{\sum_{u\in U}(R_{u,j}-\bar{R}_u)^2}}.$$

其中 $R_{u,i}$ 表示用户 u 给物品 i 的评级，$\bar{R}_u$ 表示的是用户 u 已评级项目的平均值。

- **皮尔森相关性的相似度**（Pearson correlation-based similarity）

皮尔森相关系数反映了两个变量之间的线性相关程度，它的取值在[-1, 1]之间。当两个变量的线性关系增强时，相关系数趋于 1 或-1；当一个变量增大，另一个变量也增大时，表明它们之间是正相关的，相关系数大于 0；如果一个变量增大，另一个变量却减小，表明它们之间是负相关的，相关系数小于 0；如果相关系数等于 0，表明它们之间不存在线性相关关系。皮尔森系数公式[4]如下：

$$sim(i,j)=corr_{i,j}=\frac{\sum_{u\in U}(R_{u,i}-\bar{R}_i)(R_{u,j}-\bar{R}_j)}{\sqrt{\sum_{u\in U}(R_{u,i}-\bar{R}_i)^2}\sqrt{\sum_{u\in U}(R_{u,j}-\bar{R}_j)^2}}.$$

其中，$R_{u,i}$ 表示用户 u 给物品 i 的评级，$\bar{R}_i$ 表示的是所有用户对物品 i 的评级的平均值。

除了上述三种方法，还有一些相对简单的方法，比如欧几里得距离、Jaccard 相似度（两个集合的交集元素个数 / 并集元素个数之比）。

（3）算法实现：不管电商、新闻，还是视频等多媒体应用中，一般我们还可

3 http://www10.org/cdrom/papers/519/node14.html
4 http://www10.org/cdrom/papers/519/node13.html

以借助当前的全文检索系统简化基于内容相似度推荐算法实现，比如：电商相关商品中，一般只要通过同品牌、同品类、同性别、同价位等基本属性即可快速找出相关商品；当然对于视频网站，一般我们还会经常看到同导演、同主演等相关视频，也是通过简单搜索实现的内容相似相关推荐的。不过这里为了让用户对算法理解更深刻，我们给出余弦相似度和修正余弦相似度 Java 实现代码。

```
        /**
         * 余弦相似度(cosine similarity)-CosineDistanceMeasure
         * @param src
         * @paramdest
         * @return
         */
        public double getCosineSimilar(HashMap<String,Integer> src,
HashMap<String,Integer>dest){
            double score = 0;
            if(src.size()==0 || dest.size()==0){
                return 0;
            }
            double v1=0,v2=0,fenzi=0,fmOne=0,fmTwo=0;
            Set<String>shareUserSet = new HashSet<String>();//公共元素
            for(Entry<String,Integer> item : src.entrySet()){
                v1 = item.getValue();
                fmOne += v1 * v1;
                if(dest.containsKey(item.getKey())){
                    shareUserSet.add(item.getKey());
                    v2 = dest.get(item.getKey());
                    fmTwo += v2 * v2;
                    fenzi += v1 * v2;
                }
            }
            for(Entry<String,Integer> item : dest.entrySet()){
                if(!shareUserSet.contains(item.getKey())){
                    fmTwo += item.getValue() * item.getValue();
                }
            }
            shareUserSet.clear();
            shareUserSet = null;
            if(fmOne==0 || fmTwo==0){
                return 0;
            }else{
                score = fenzi / Math.sqrt(fmOne * fmTwo);
            }
            return score;
        }

        /**
         * 调整余弦相似度-Adjusted Cosine Similarity
         * @param src
         * @paramdest
```

```
         * @return
         */
        public double getAdjustCosineSimilar(Map<String,Integer> src,
Map<String,Integer>dest){
            double score = 0;
            int srcNum = src.size();
            int destNum = dest.size();
            if(srcNum==0 || destNum==0){
                return 0;
            }
            double v1=0,v2=0,axv=0,ayv=0,fenzi=0,fmOne=0,fmTwo=0;
            for(int s : src.values()){
                axv += s;
            }
            axv = axv/srcNum;  //src用户评分平均值
            for(int d : dest.values()){
                ayv += d;
            }
            ayv = ayv/destNum;  //dest用户评分平均值
            Set<String>shareUserSet = new HashSet<String>();
            for(Entry<String,Integer> item : src.entrySet()){
                v1 = item.getValue() - axv;
                fmOne += v1 * v1;
                if(dest.containsKey(item.getKey())){
                    shareUserSet.add(item.getKey());
                    v2 =dest.get(item.getKey()) - ayv;
                    fmTwo += v2 * v2;
                    fenzi += v1 * v2;
                }
            }
            double tmpScore = 0;
            for(Entry<String,Integer> item : dest.entrySet()){
                if(!shareUserSet.contains(item.getKey())){
                    tmpScore = item.getValue() - ayv;
                    fmTwo += tmpScore * tmpScore;
                }
            }
            double shareNum = shareUserSet.size() * 1.0;
            shareUserSet.clear();
            shareUserSet = null;
            if(fmOne==0 || fmTwo==0){
                return 0;
            }else{
                score = (fenzi/Math.sqrt(fmOne * fmTwo)) *
Math.sqrt(shareNum/src.size());
            }
            return score;
        }
```

上述两例之所以都使用 Map 结构表示向量，是因为，生产环境下 VSM 的向量大多是稀疏的，如果向量是非稀疏的，则可以考虑借助 JDK8 产生的 Optional

容器一一对应数组化。

2. 关联规则

（1）使用场景

数据挖掘中的经典案例“尿布与啤酒”就是关联规则的典型体现。关联规则最初提出的动机是针对购物篮分析（Market Basket Analysis）问题提出的。假设分店经理想更多地了解顾客的购物习惯。特别是，想知道哪些商品顾客可能会在一次购物时同时购买？为回答该问题，可以对商店的顾客事物零售数量进行购物篮分析。该过程通过发现顾客放入“购物篮”中的不同商品之间的关联，分析顾客的购物习惯。这种关联的发现可以帮助零售商了解哪些商品频繁地被顾客同时购买，从而帮助他们开发更好的营销策略。

此外，当我们浏览一个电商网站时，在商品详情页我们经常看到：

看了这个商品的用户**%还看了
买了这个商品的用户**%还买了

当我们浏览一个视频网站时，当我们观看了一个视频后，视频详情页经常看到：

看了此片的用户**%还看了

以上例子都是关联规则的典型使用场景，大多数人看到这个可能会有所触动。

（2）算法介绍

关联规则是形如 $X \rightarrow Y$ 的蕴涵式，其中，X 和 Y 分别称为关联规则的先导（antecedent 或 left-hand-side，LHS）和后继（consequent 或 right-hand-side，RHS）。其中，关联规则 XY，存在支持度和信任度和提升度几个概念。

- 支持度（Support），表示项集$\{X, Y\}$在总项集里出现的概率[5]。公式为：

5 关联分析中的支持度、置信度和提升度 http://blog.sina.com.cn/s/blog_5df073190102vxcq.html

$$\text{Support}(X \rightarrow Y) = P(X,Y) / P(\text{I}) = P(X \cup Y) / P(\text{I}) = \text{num}(X \cup Y) / \text{num}(I)$$

其中，I 表示总事务集。num()表示求事务集里特定项集出现的次数。比如，num(I)表示总事务集的个数，num($X \cup Y$)表示含有{X,Y}的事务集的个数（个数也叫次数）。

- 置信度（Confidence），置信度表示在先决条件 X 发生的情况下，由关联规则“$X \rightarrow Y$”推出 Y 的概率。即在含有 X 的项集中，含有 Y 的可能性，公式为：

$$\text{Confidence}(X \rightarrow Y) = P(Y|X) \quad = P(X,Y) / P(X) = P(X \cup Y) / P(X)$$

- 提升度（Lift），提升度表示含有 X 的条件下，同时含有 Y 的概率，与不含 X 却含 Y 的条件下的概率之比。公式为：

$$\text{Lift}(X \rightarrow Y) = \text{P}(Y|X) / \text{P}(Y)$$

满足最小支持度和最小置信度的规则，叫做“强关联规则”。然而，强关联规则里，也分有效的强关联规则和无效的强关联规则。

如果 Lift($X \rightarrow Y$)>1，则规则“$X \rightarrow Y$”是有效的强关联规则。

如果 Lift($X \rightarrow Y$) <=1，则规则“$X \rightarrow Y$”是无效的强关联规则。

特别地，如果 Lift($X \rightarrow Y$) =1，则表示 X 与 Y 相互独立。

（3）关联规则的挖掘过程

关联规则的挖掘过程主要包含两个阶段：第一阶段必须先从资料集合中找出所有的高频项目组（Frequent Itemsets），第二阶段再由这些高频项目组中产生关联规则（Association Rules）。

第一阶段必须从原始资料集合中，找出所有高频项目组（Large Itemsets）。高频的意思是指某一项目组出现的频率相对于所有记录而言，必须达到某一水平。一项目组出现的频率称为支持度（Support），以一个包含 A 与 B 两个项目的

2-itemset 为例，我们可以经由公式（1）求得包含{A,B}项目组的支持度，若支持度大于等于所设定的最小支持度（Minimum Support）门槛值时，则{A,B}称为高频项目组。一个满足最小支持度的 k-itemset，则称为高频 k-项目组（Frequent k-itemset），一般表示为 Large k 或 Frequent k。算法并从 Large k 的项目组中再产生 Large k+1，直到无法再找到更长的高频项目组为止[6]。

第二阶段是要产生关联规则（Association Rules）。从高频项目组产生关联规则，是利用前一步骤的高频 k-项目组来产生规则，在最小信赖度（Minimum Confidence）的条件门槛下，若一规则所求得的信赖度满足最小信赖度，称此规则为关联规则。例如：经由高频 k-项目组{A,B}所产生的规则 AB，其信赖度可经由公式（2）求得，若信赖度大于等于最小信赖度，则称 AB 为关联规则。

（4）相关算法

- Apriori 算法

使用候选项集找频繁项集。Apriori 算法是一种最有影响的挖掘布尔关联规则频繁项集的算法。其核心是基于两阶段频集思想的递推算法。该关联规则在分类上属于单维、单层、布尔关联规则。在这里，所有支持度大于最小支持度的项集称为频繁项集，简称频集。

该算法的基本思想是：首先找出所有的频集，这些项集出现的频繁性至少和预定义的最小支持度一样。然后由频集产生强关联规则，这些规则必须满足最小支持度和最小可信度。然后使用第 1 步找到的频集产生期望的规则，产生只包含集合的项的所有规则，其中每一条规则的右部只有一项，这里采用的是中规则的定义。一旦这些规则被生成，那么只有那些大于用户给定的最小可信度的规则才被留下来。为了生成所有频集，使用了递推的方法。

Apriori 算法采用了逐层搜索的迭代的方法，算法简单明了，没有复杂的理论推导，也易于实现。但其有一些难以克服的缺点：对数据库的扫描次数过多；会

6 关联规则百度百科 http://baike.baidu.com/view/1076817.htm

产生大量的中间项集；采用唯一支持度；算法的适应面窄等。

- 基于划分的算法

这个算法先把数据库从逻辑上分成几个互不相交的块，每次单独考虑一个分块并对它生成所有的频集，然后把产生的频集合并，用来生成所有可能的频集，最后计算这些项集的支持度。这里分块的大小选择要使得每个分块可以被放入主存，每个阶段只需被扫描一次。而算法的正确性是由每一个可能的频集至少在某一个分块中是频集保证的。该算法是可以高度并行的，可以把每一分块分别分配给某一个处理器生成频集。产生频集的每一个循环结束后，处理器之间进行通信来产生全局的候选 k-项集。通常这里的通信过程是算法执行时间的主要瓶颈；而另一方面，每个独立的处理器生成频集的时间也是一个瓶颈。

- FP-树频集算法

针对 Apriori 算法的固有缺陷，J. Han 等提出了不产生候选挖掘频繁项集的方法，即 FP-树频集算法。采用分而治之的策略，在经过第一遍扫描之后，把数据库中的频集压缩进一棵频繁模式树（FP-tree），同时依然保留其中的关联信息，随后再将 FP-tree 分化成一些条件库，每个库和一个长度为 1 的频集相关，然后再对这些条件库分别进行挖掘。当原始数据量很大的时候，也可以结合划分的方法，使得一个 FP-tree 可以放入主存中。实验表明，FP-growth 对不同长度的规则都有很好的适应性，同时在效率上较之 Apriori 算法有巨大的提高。

3. 协同过滤

（1）基本思想

集体智慧（Collective Intelligence）是指在大量人群的行为和数据中收集答案，帮助你对整个人群得到统计意义上的结论，这些结论是我们在单个个体上无法得到的，它往往是某种趋势或者人群中共性的部分。在当今移动互联网时代，可利用集体智慧构建更加有趣的应用或者得到更好的用户体验。

协同过滤是利用集体智慧的一个典型方法[7]。要理解什么是协同过滤*(Collaborative Filtering，简称 CF)*，*首先想一个简单的问题，如果你现在想看个*电影，但你不知道具体看哪部，你会怎么做？大部分的人会问问周围的朋友，看看最近有什么好看的电影推荐，而我们一般更倾向于从口味比较类似的朋友那里得到推荐。这就是协同过滤的核心思想。换句话说，就是借鉴和你相关人群的观点来进行推荐，很好理解。

俗话说“物以类聚、人以群分”，拿看电影这个例子来说，如果你喜欢《蜘蛛侠》《功夫之王》《李小龙传奇》《武林》等电影，另外有个人也都喜欢这些电影，而且他还喜欢《少林足球》，则很有可能你也喜欢《少林足球》这部电影。所以说，当一个用户 A 需要个性化推荐时，可以先找到和他兴趣相似的用户群体 G，然后把 G 喜欢的、并且 A 没有听说过的物品推荐给 A，这就是基于用户的系统过滤算法。

(2) 核心原理及实现步骤

要实现协同过滤，需要进行如下几个步骤：

收集用户偏好——找到相似的用户或者物品——计算并推荐。

1) 收集用户偏好

从用户的行为和偏好中发现规律，并基于此进行推荐，所以如何收集用户的偏好信息成为系统推荐效果最基础的决定因素。用户有很多种方式向系统提供自己的偏好信息，比如：评分、投票、转发、保存书签、购买、点击流、页面停留时间等等。

以上的用户行为都是通用的，在实际推荐引擎设计中可以自己多添加一些特定的用户行为，并用它们表示用户对物品的喜好程度。通常情况下，在一个推荐系统中，用户行为都会多于一种，那么如何组合这些不同的用户行为呢？基本上

7 探索推荐引擎内部的秘密，第 2 部分：深入推荐引擎相关算法 - 协同过滤--
http://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy2/

有如下两种方式：

- 将不同的行为分组

一般可以分为查看和购买，然后基于不同的用户行为，计算不同用户或者物品的相似度。类似与当当网或者亚马逊给出的“购买了该书的人还买了”，“查看了该书的人还看了”等。

- 不同行为产生的用户喜好对它们进行加权

对不同行为产生的用户喜好进行加权，然后求出用户对物品的总体喜好。

2）找到相似的用户或者物品

对用户的行为分析得到用户的喜好后，可以根据用户的喜好计算相似用户和物品，然后基于相似用户或物品进行推荐。这就是协同过滤中的两个分支了，基于用户的和基于物品的协同过滤。

在推荐的场景中，在用户-物品偏好的二维矩阵中，我们可以将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，或者将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度。关于相似度的计算有很多种方法，比如常用的余弦夹角、欧几里得距离度量、皮尔逊相关系数等，上节内容相似度算法已经介绍过。这里补充一下前面没有介绍的。

欧几里得距离：最初用于计算欧几里得空间中两个点的距离，假设 x，y 是 n 维空间的两个点，它们之间的欧几里得距离是 d(x,y)，当用欧几里得距离表示相似度，一般采用以下公式 sim(x,y)进行转换：距离越小，相似度越大。

$$d(x,y)=\sqrt{(\sum(x_i-y_i)^2)} \qquad sim(x,y)=\frac{1}{1+d(x,y)}$$

Tanimoto 系数也称为 **Jaccard 系数**，是 Cosine 相似度的扩展，也多用于计算文档数据的相似度：

$$T(x,y)=\frac{x\bullet y}{\|x\|^2+\|y\|^2-x\bullet y}=\frac{\sum xiyi}{\sqrt{\sum xi^2}+\sqrt{\sum yi^2}-\sum xiyi}$$

3）核心原理

基于用户的 CF 的基本思想相当简单，基于用户对物品的偏好找到相邻邻居用户，然后将邻居用户喜欢的推荐给当前用户。计算上，就是将一个用户对所有物品的偏好作为一个向量来计算用户之间的相似度，找到 K 邻居后，根据邻居的相似度权重，以及他们对物品的偏好，预测当前用户没有偏好的未涉及物品，计算得到一个排序的物品列表作为推荐。图 9.2 给出了一个例子，对于用户 A，根据用户的历史偏好，这里只计算得到一个邻居-用户 C，然后将用户 C 喜欢的物品 D 推荐给用户 A。

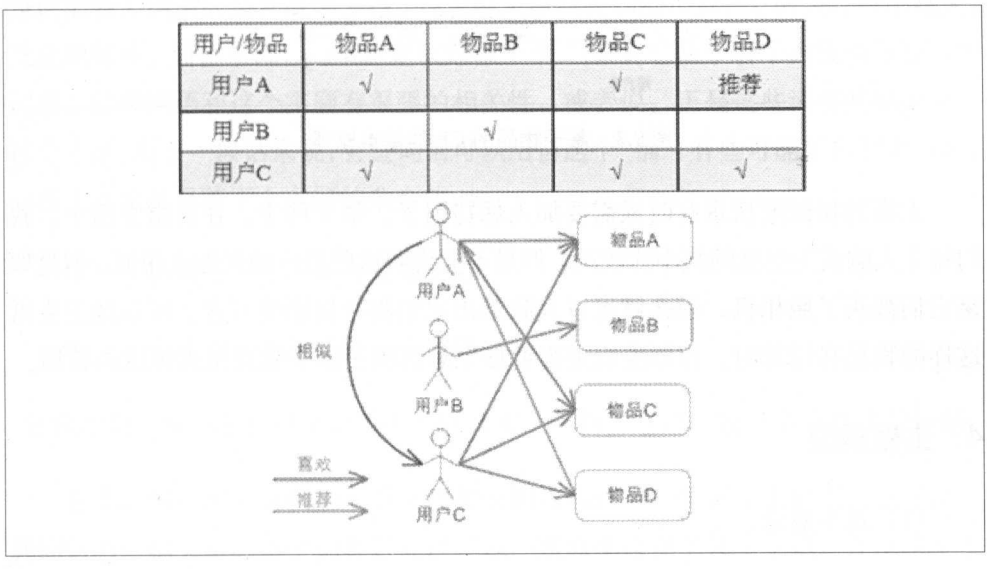


图 9.2 基于用户的 CF 的基本原理

基于物品的 CF 的原理和基于用户的 CF 类似，只是在计算邻居时采用物品本身，而不是从用户的角度，即基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。从计算的角度看，就是将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的

物品列表作为推荐。图 9.3 给出了一个例子，对于物品 A，根据所有用户的历史偏好，喜欢物品 A 的用户都喜欢物品 C，得出物品 A 和物品 C 比较相似，而用户 C 喜欢物品 A，那么可以推断出用户 C 可能也喜欢物品 C。

用户/物品	物品A	物品B	物品C
用户A	√		√
用户B	√	√	√
用户C	√		推荐

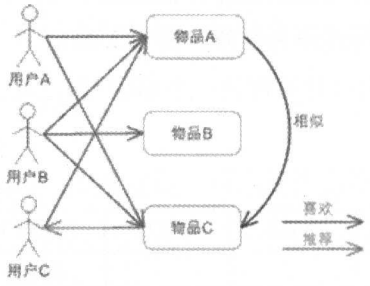


图 9.3　基于物品的 CF 的基本原理

上面的相似度权重有时候需要加入惩罚因子，举个例子，在日常生活中，我们每个人购买卫生纸的频率比较高，但是不能说明这些用户的兴趣点相似，但是如果它们都买了照相机，那么就可以大致推出它们都是摄影爱好者。所以像卫生纸这样的物品在计算时，相似度权重需要加上惩罚因子或干脆直接去掉这类数据。

4. 主题模型

（1）基本概念

传统判断两个文档相似性的方法是通过查看两个文档共同出现单词的多少，如 TF-IDF 等，这种方法没有考虑到文字背后的语义关联，可能在两个文档共同出现的单词很少甚至没有，但两个文档是相似的。例如，有两个句子分别如下：

“乔布斯离我们而去了。”
“苹果价格会不会降？”

可以看到上面这两个句子没有共同出现的单词，但这两个句子是相似的，如果按传统的方法判断这两个句子肯定不相似，所以在判断文档相关性的时候需要考虑到文档的语义，而语义挖掘的利器是主题模型。主题模型，顾名思义，就是对文字中隐含主题的一种建模方法。还是上面的例子，“苹果”这个词的背后既包含是苹果公司这样一个主题，也包括了水果的主题。当我们和第一句进行比较时，苹果公司这个主题就和“乔布斯”所代表的主题匹配上了，因而我们认为它们是相关的。

在这里，我们先定义一下主题究竟是什么[8]。主题就是一个概念、一个方面。它表现为一系列相关的词语。比如一个文章如果涉及“百度”这个主题，那么“中文搜索”、“李彦宏”等词语就会以较高的频率出现，而如果涉及“IBM”这个主题，那么“笔记本”等就会出现得很频繁。如果用数学来描述一下的话，**主题就是词汇表上词语的条件概率分布**。与主题关系越密切的词语，它的条件概率越大，反之则越小。通俗来说，一个主题就好像一个“桶”，它装了若干出现概率较高的词语。这些词语和这个主题有很强的相关性，或者说，正是这些词语共同定义了这个主题。对于一段话来说，有些词语可以出自这个“桶”，有些可能来自那个“桶”，一段文本往往是若干个主题的杂合体。

(2) 主题模型工作原理及发展历程

- 工作原理，主题模型生成过程

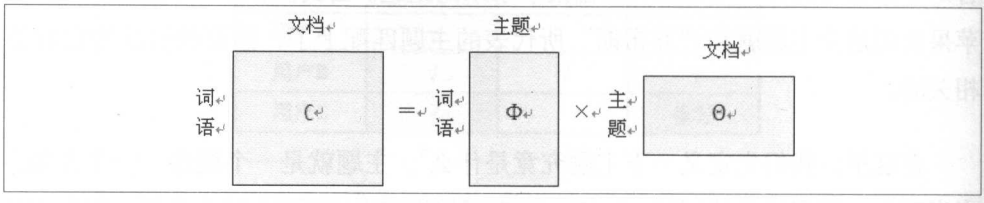
怎样才能生成主题？对文章的主题应该怎么分析？这是主题模型要解决的问题。

首先，可以用生成模型来看文档和主题这两件事。所谓生成模型，就是说，我们认为一篇文章的每个词都是通过“以一定概率选择了某个主题，并从这个主题中以一定概率选择某个词语”这样一个过程得到的。那么，如果我们要生成一篇文档，它里面的每个词语出现的概率为：

8 主题模型 http://www.cnblogs.com/lixiaolun/p/4455764.html

$$p(词语|文档)=\sum_{主题} p(词语|主题)\times p(主题|文档)$$

这个概率公式可以用矩阵表示：



其中“文档-词语”矩阵表示每个文档中每个单词的词频，即出现的概率；“主题-词语”矩阵表示每个主题中每个单词的出现概率；“文档-主题”矩阵表示每个文档中每个主题出现的概率。

假如我们有很多的文档，比如大量的网页，先对所有文档进行分词，得到一个词汇列表。这样每篇文档就可以表示为一个词语的集合。对于每个词语，我们可以用它在文档中出现的次数除以文档中词语的数目作为它在文档中出现的概率。这样，对任意一篇文档，左边的矩阵是已知的，右边的两个矩阵未知。而主题模型就是用大量已知的“词语－文档”矩阵，通过一系列的训练，推理出右边的“词语－主题”矩阵Φ和“主题文档”矩阵Θ。

- 发展历程

主题模型是一种对文字隐含主题进行挖掘建模的方法，它的主要思想是：一篇文本是由若干个不同领域或方面的主题构成的，而主题则可以理解成多个词语的一种概率分布。因此，主题模型是文本的一种生成模型，它通过将文本和词语的维度转化成文本与主题、主题与词语的维度，通过将文本映射到主题空间，即认为一个文本由若干个主题随机组成，从而捕获各个文本之间潜在的语义关系。

潜在语义分析打破了以往人们对于文本表示的思维定式[9]：文本是表示在词典

9 基于主题模型的个性化新闻推荐系统的研究与实现 http://www.doc88.com/p-1991926652904.html

空间上的。潜在语义分析创新性引入了语义维度，语义维度是文本集上信息的浓缩表示，而文本则是在这些语义维度上的一个表示。形象地说，在以往的文本－词语映射表示种，引入了语义维度，即文本－语义－词语，实现了文本在语义空间上的低维表示。低维表示或降维是传统数据分析中的一个重要手段，其目的就是想要找到一个对于数据更具有表现力、更压缩的表示方法。低维表示的好处在于能够去除噪音影响、降低数据表示成本等。

LSA 通过引入语义维度解决了上文描述的词语的多义性问题，而随着概率统计分析在文本建模应用的不断发展，潜在语义分析从线性代数的分析模式（SVD 分解）中被进一步提升到概率统计的分析模式，即 pLSA（probabilistic Latent Semantic Analysis），其使用概率模型模拟潜在的语义空间，在 LSA 中每个语义维度对应一个特征向量，而在概率模型中，每个语义维度则对应到一个词典 V 的概率分布。

pLSA 在 LSA 的基础上进行概率拓展，可以引入先验信息并且更方便地对模型进行拓展，它使得很多启发式处理手段可以得到理论上的解释。pLSA 主要使用的是 EM（期望最大化）算法，包含两个不断迭代的过程：E（期望）过程和 M（最大化）过程。用一个形象的例子来说吧：比如说食堂的大师傅炒了一盘菜，要等分成两份给两个人吃，显然没有必要拿天平秤去一点点地精确称量，最简单的办法是先随意地把菜分到两个碗中，然后观察是否一样多，把比较多的那一份取出一点放到另一个碗中，这个过程一直重复下去，直到大家看不出两个碗里的菜量有什么差别为止。对于主题模型训练来说，“计算每个主题里的词语分布”和“计算训练文档中的主题分布”就好比是在往两个人碗里分菜。在 E 过程中，我们通过贝叶斯公式可以由“词语－主题”矩阵计算出“主题－文档”矩阵。在 M 过程中，我们再用“主题－文档”矩阵重新计算“词语－主题”矩阵。这个过程一直这样迭代下去。EM 算法的神奇之处就在于它可以保证这个迭代过程是收敛的。也就是说，我们在反复迭代之后，就一定可以得到趋向于真实值的 Φ 和 Θ。

pLSA 并不是一个“完整的”贝叶斯模型，原因是在 pLSA 中，仍然将文本－主题的分布和主题－词语的分布看做是参数而不是随机变量，因此在实际应用中因为其准确率及计算量有很大的局限性。

“主题模型”在“Latent Dirichlet Allocation（LDA，潜在狄利克雷分布）”中第一次被提出，即之前潜在语义分析中的语义维度。主题是语料依赖的，也就是对于不同的语料集合，它们背后隐藏的语义维度各不相同。主题是对语料集合上的高度抽象、压缩表示。LDA 在 pLSA 基础上进行贝叶斯化，使得参数具备了概率分布，变成了随机变量，从而有效解决了 pLSA 存在的问题，并广泛地应用在情感分析、分布挖掘和网络信息挖掘领域中。

（3）相关算法

1）基于 LDA 主题模型

LDA 是一个三层贝叶斯概率模型，包含词、主题和文档三层结构。LDA 是产生式全概率生成模型，是典型的有向概率图模型。给定一个文档集合，LDA 将每个文档表示为一个主题混合，而每个主题是固定词表上的一个多项式分布。文档到主题服从 Dirichlet 分布，主题到词服从多项式分布。每个文档有一个特定的主题比例，从 Dirichlet 分布中抽样产生；主题之间是相互独立的。文本被看成是有多个浅层的主题混合组成的。这些主题被集合中的所有文档所共享，而每个文档有一个特定的主题分布。LDA 主题模型是一种统计语言模型，通过求出每个词对应的隐含主题，利用 Dirichlet 先验得到主题分布，反复抽样产生文档的每一个词。

图 9.4 中，α 为主题分布的 Dirichlet 超参数，β 为词分布的 Dirichlet 超参数，N 为每个文档中词的个数，M 为文档个数。

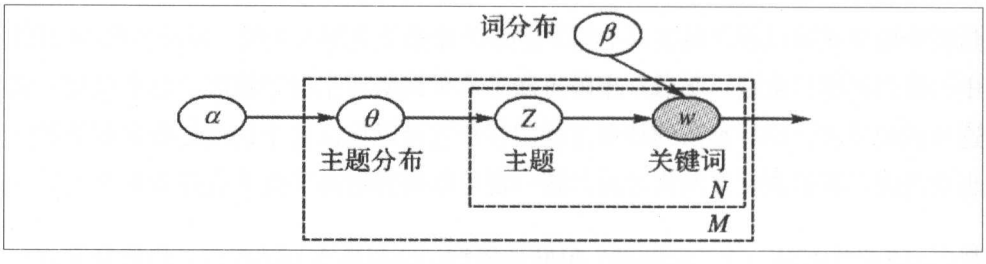


图 9.4　LDA 主题模型

图 9.4 中，α 为主题分布的 Dirichlet 超参数，β 为词分布的 Dirichlet 超参数，

N 为每个文档中词的个数，M 为文档个数。文档生成过程如下：1）根据 α 从主题的多项分布中抽取一个主题；2）根据 β 从词的多项分布中抽取这个主题对应的一个词；3）重复第 1）、2），直到抽取出文章中所有的词。这种方法首先选定一个主题向量 $\boldsymbol{\theta}$，确定每个主题被选择的概率。然后在生成每个单词的时候，从主题分布向量 $\boldsymbol{\theta}$ 中选择一个主题 z，按主题 z 的单词概率分布生成一个单词。

LDA 模型是当前最具有代表性：也是最流行的一种概率主题模型。在文本挖掘、知识发现、话题跟踪，以及多文档摘要等领域得到了广泛的良好应用。LDA 是一种非监督机器学习技术，可以用来识别大规模文档集或语料库中潜藏的主题信息。LDA 模型从一个数据集合中挖掘指定个数的潜在主题模型，通过这些主题模型表示一个文本，从而达到特征降维的目的，LDA 模型的同一个主题中的特征通常比较相关或相近，存在如高共现度、高相关度等的联系。

2）基于聚类主题模型

文本聚类是文本挖掘和信息组织导航的重要手段和方法，它把一个文本集分成若干称为簇（Cluster）的子集，每个簇的文本之间具有较大的相似性，而簇间的文本具有较小的相似性。作为一种无监督的机器学习方法，聚类由于不需要训练过程，以及不需要预先对文档手工标注类别，因此具有较高的灵活性和自动化处理能力，成为对文本信息进行有效组织、摘要和导航的重要手段。

聚类算法有很多种，这里重点给大家介绍两种比较常用的聚类算法[10]：划分聚类和层次聚类。

- 基于划分的聚类算法

主要有 K-means 算法、K-均值算法等，该方法要求将文档集水平分割成 K 个类（$k<<N$），且每个类均最低包含一个文档。K-means 算法是一种基于质心的贪心算法，其原理是首先首先选择 K 个初始文档作为聚类中心点，后以聚簇中的文档均值作为判定标准，将剩余文档一次归类并更新聚簇均值，重复迭代直至聚簇划

10 基于 LDA 模型的文本聚类研究 http://www.doc88.com/p-919951770055.html

分不再改变。

若定义 K 为类别数目，L 为聚类过程中迭代的次数、N 为文档集规模，则 K-means 的算法复杂度为 $O(KLN)$。可以看出，其优点在于：计算复杂度低、适合大规模文本集操作。缺点则是需手动制定 K 值、容易受噪声及孤立点影响及优化的局限性。

- 基于层次的聚类算法

将文档对象组织成一颗聚类树，生成树中包含了类间的层次信息及文档间的相似度，根据自底向上或自顶向下的聚合顺序分为凝聚（Aggregation）层次聚类、分裂（Division）层次聚类。

凝聚层次聚类的思想是首先将每个文档均当做原子聚类点，预定文本相似度的阀值，后依次按照阀值合并原子类，直到类合并到最上层或者满足终止条件退出。分裂层次聚类则与之相反，事先将整个文档集作为一个大类，后根据文本相似度的阀值进行逐步细分成小类，直到均自成一类或满足终止条件。在基于层次聚类算法中，类的合并或分裂军需计算类与类之间的距离，目前主要有以下四种距离度量方法：

平均距离：	$d_{avg}(c_i,c_j)=\frac{1}{n_i n_j}\sum_{p\in c_i}\sum_{p'\in c_j}\lvert p-p'\rvert$
平均值距离：	$d_{mean}(c_i,c_j)=\lvert m_i-m_j\rvert$
最大距离：	$d_{max}(c_i,c_j)=\max_{p\in c_i,p'\in c_j}\lvert p-p'\rvert$
最小距离：	$d_{min}(c_i,c_j)=\min_{p\in c_i,p'\in c_j}\lvert p-p'\rvert$

其中 n 是聚簇 c 中的对象个数，m 是 c 中距离平均值，| p-p' | 则是文本对象 p 与 p' 间的距离，可以看出，层次聚类算法的优点在于：无时限设参；适用于不规则形状的类；聚类粒度可控。同时不足在于：时间复杂度高达 $O(n*n)$；制止条件不确定；重构不容易、无法回溯；不适合动态数据集。

和其他大多数自然语言处理面临的问题一样，文本聚类也是和语义密切相关的。然而传统上大多数文本聚类方法都只是利用词频统计信息来进行文本相似度

度量的评估，人们总是期望聚类算法能深入到概念级和语义级，在以往的文本聚类中，文本表示通常选择向量空间模型（Vector Space Model，VSM）算法，选择词作为特征项，将文档集构造为一个高维、稀疏的词条-文本矩阵。然而向量空间模型只是词面上的匹配，不能从语义上理解文档之间的关系。另一种比较流行的方法就是通过主题模型来给文本建模，将文本表示成主题按一定比例的混合，这样能够充分挖掘文本集合的内在信息。

3）基于 Word2vector

Word2vector 是 Google 在 2013 年年中开源的一款将词表征为实数值向量的高效工具[11]，其利用深度学习的思想，可以通过训练，把对文本内容的处理简化为 K 维向量空间中的向量运算，而向量空间上的相似度可以用来表示文本语义上的相似度。Word2vector 输出的词向量可以被用来做很多 NLP 相关的工作，比如聚类、找同义词、词性分析等等。如果换个思路，把词当做特征，那么 Word2vector 就可以把特征映射到 K 维向量空间，可以为文本数据寻求更加深层次的特征表示。

Word2vector 使用的是 Distributed representation 的词向量表示方式。Distributed representation 最早由 Hinton 在 1986 年提出。其基本思想是通过训练将每个词映射成 K 维实数向量（K 一般为模型中的超参数），通过词之间的距离（比如 cosine 相似度、欧氏距离等）来判断它们之间的语义相似度，其采用一个三层的神经网络：输入层-隐层-输出层。有个核心的技术是根据词频用 Huffman 编码，使得所有词频相似的词隐藏层激活的内容基本一致，出现频率越高的词语，被激活的隐藏层数目越少，这样有效地降低了计算的复杂度。而 Word2vector 大受欢迎的一个原因正是其高效性，Mikolov 在其论文中指出，一个优化的单机版本一天可训练上千亿词。

这个三层神经网络本身是对语言模型进行建模，但也同时获得一种单词在向量空间上的表示，而这个副作用才是 Word2vec 的真正目标。

与潜在语义分析（Latent Semantic Index, LSI）、潜在狄立克雷分配（Latent

11 文本深度表示模型 Word2Vec http://wei-li.cnblogs.com/p/word2vec.html

Dirichlet Allocation，LDA）的经典过程相比，Word2vector 利用了词的上下文，语义信息更加丰富。

Word2vec 官方地址：https://code.google.com/p/word2vec/，国内也有对此研究的同行，开发了 Java 版：https://github.com/NLPchina/Word2VEC_java。

5．逻辑回归（Logistic Regression, LR）

（1）LR 的基本概念

LR 是一种广义线性回归（generalized linear model），因此与多重线性回归分析有很多相同之处。它们的模型形式基本上相同，都具有 $w'x+b$，其中 w 和 b 是待求参数，其区别在于它们的因变量不同，多重线性回归直接将 $w'x+b$ 作为因变量，即 $y=w'x+b$，而 logistic 回归则通过函数 L 将 $w'x+b$ 对应一个隐状态 p，$p=L(w'x+b)$，然后根据 p 与 $1-p$ 的大小决定因变量的值。如果 L 是 logistic 函数，就是 logistic 回归，如果 L 多项式函数就是多项式回归。LR 的因变量可以是二分类的，也可以是多分类的，但是二分类的更为常用，也更加容易解释。所以实际中最常用的就是二分类的逻辑回归。

LR 模型其实仅在线性回归的基础上，套用了一个逻辑函数，但也就由于这个逻辑函数，使得逻辑回归模型成为了机器学习领域一颗耀眼的明星，在互联网领域得到了广泛的应用，无论是在广告系统中进行 CTR 预估，推荐系统中的预估转换率，反垃圾系统中的识别垃圾内容等都可以看到它的身影。LR 以其简单的原理和应用的普适性受到了广大应用者的青睐。

（2）适用条件及原理

逻辑回归模型的适用条件如下：

（1）因变量为二分类的分类变量或某事件的发生率，并且是数值型变量。但是需要注意，重复计数现象指标不适用于 Logistic 回归。

（2）残差和因变量都要服从二项分布。二项分布对应是分类变量，所以不是

正态分布，进而不是用最小二乘法，而是最大似然法来解决方程估计和检验问题。

（3）自变量和 Logistic 概率是线性关系。

（4）各观测对象间相互独立。

逻辑回归的原理：如果直接将线性回归的模型套用到 Logistic 回归中，会造成方程二边取值区间不同和普遍的非直线关系。因为 Logistic 中因变量为二分类变量，某个概率作为方程的因变量估计值取值范围为 0~1，但是，方程右边取值范围是无穷大或者无穷小。所以，才引入 LR。LR 的实质是发生概率除以没有发生概率再取对数。就是这个不太繁琐的变换改变了取值区间的矛盾和因变量自变量间的曲线关系。究其原因，是发生和未发生的概率成为了比值，这个比值就是一个缓冲，将取值范围扩大，再进行对数变换，整个因变量改变。不仅如此，这种变换往往使得因变量和自变量之间呈线性关系，这是根据大量实践而总结的。所以，Logistic 回归从根本上解决因变量要不是连续变量怎么办的问题。还有，Logistic 应用广泛的原因是许多现实问题跟它的模型吻合。例如一件事情是否发生跟其他数值型自变量的关系。

3．LR基本模型的求解方法

LR 模型中，通过特征权重向量对特征向量的不同维度上的取值进行加权，并用逻辑函数将其压缩到 0~1 的范围，作为该样本为正样本的概率，逻辑函数[12]为 $\sigma(x)=\frac{1}{1+e^{-x}}$。

给定 M 个训练样本 $(X_1,y_1),(X_2,y_2)....(X_M,y_M)$，其中 $X_j=\{x_{ji}|i=1,2,...N\}$ 为 N 维的实数向量（特征向量，这里所有向量不作说明都为列向量）；y_j 取值为+1 或-1，为分类标签，+1 表示样本为正样本，-1 表示样本为负样本。在 LR 模型中，第 j 个样本为正样本的概率是：

$$P(y_j=1|W,X_j)=\frac{1}{1+e^{-W^TX_j}}$$

12 详解并行逻辑回归 http://www.csdn.net/article/2014-02-13/2818400-2014-02-13

其中 W 是 N 维的特征权重向量，也就是 LR 问题中要求解的模型参数。

求解 LR 问题，就是寻找一个合适的特征权重向量 W，使得对于训练集里面的正样本，$P(y_j = 1|W, X_j)$值尽量大；对于训练集里面的负样本，这个值尽量小（或 $P(y_j = -1|W^T, X_j)$尽量大）。用联合概率来表示：

$$\max_W p(W) = \prod_{j=1}^{M} \frac{1}{1 + e^{-y_j W^T X_j}}$$，对上式求 log 并取负号，则等价于：

$$\min_W f(W) = \sum_{j=1}^{M} log\,(1 + e^{-y_j W^T X_j})$$，上述公式就是 LR 求解的目标函数。图 9.5 演示了求解最优化目标函数的基本步骤。

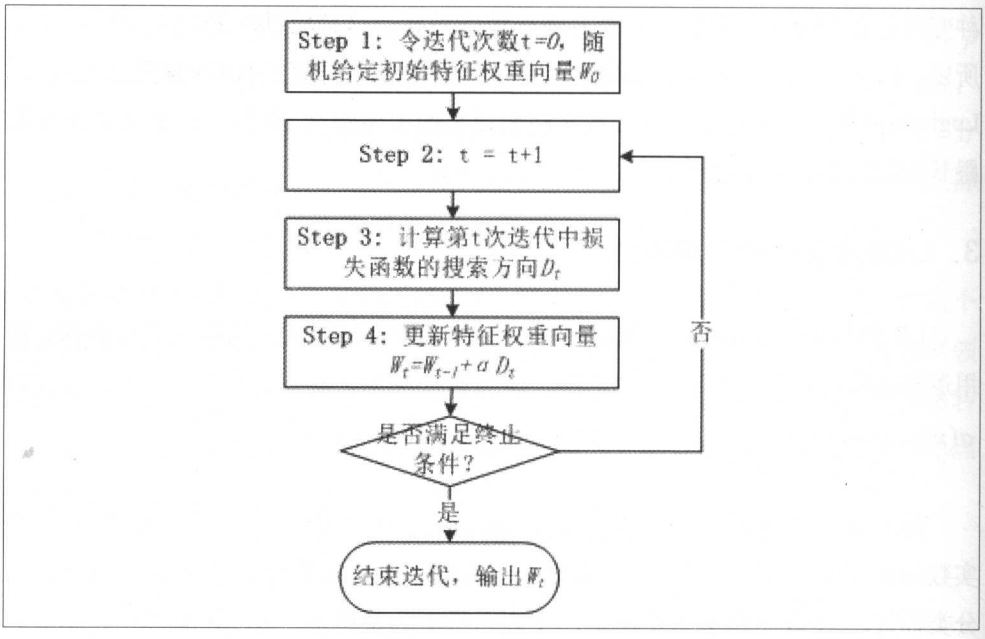


图 9.5　求解最优化目标函数的基本步骤

寻找合适的 W 令目标函数 $f(W)$最小，是一个无约束最优化问题，解决这个问题的通用做法是随机给定一个初始的 W0，通过迭代，在每次迭代中计算目标函数的下降方向并更新 W，直到目标函数稳定在最小的点。

不同的优化算法的区别就在于目标函数下降方向 D_t 的计算。下降方向是通过对目标函数在当前的 W 下求一阶倒数（梯度，Gradient）和求二阶导数（海森矩阵，Hessian Matrix）得到。常见的算法有梯度下降法、牛顿法、拟牛顿法。

9.1.4 用户画像数据仓库

1. 用户画像（User Profile）定义

在互联网逐渐步入大数据时代后，不可避免地给企业及消费者行为带来一系列改变与重塑。其中最大的变化莫过于，消费者的一切行为在企业面前似乎都将是“可视化”的。随着大数据技术的深入研究与应用，企业的专注点日益聚焦于怎样利用大数据来为精准营销服务，进而深入挖掘潜在的商业价值。于是，“用户画像”的概念也就应运而生，作为大数据的根基，它完美地抽象出一个用户的信息全貌，为进一步精准、快速地分析用户行为习惯、消费习惯等重要信息，提供了足够的数据基础，奠定了大数据时代的基石。

用户画像，即用户信息标签化[13]，就是企业通过收集与分析消费者社会属性、生活习惯、消费行为等主要信息的数据之后，完美地抽象出一个用户的商业全貌，可看成是企业应用大数据技术的基本方式。用户画像为企业提供了足够的信息基础，能够帮助企业快速找到精准用户群体，以及用户需求等更为广泛的反馈信息。

用户画像的焦点工作就是为用户打“标签”，而一个标签通常是人为规定的高度精练的特征标识，如年龄、性别、地域、用户偏好等，最后将用户的所有标签综合来看，就可以勾勒出该用户的立体“画像”了。用户画像的目标是通过分析用户行为，最终为每个用户打上标签，以及该标签的权重。其中：

- 标签，表征了内容，用户对该内容有兴趣、偏好、需求等。
- 权重，表征了指数，用户的兴趣、偏好指数，也可能表征用户的需求度，可以简单地理解为可信度，概率。

13 你确定你真的懂用户画像 http://www.woshipm.com/it/250043.html/comment-page-1

2．用户画像来源UGC、PGC和OGC

当前移动网络时代，用户画像出自社交网络媒体内容，而社交网络媒体内容来源主要有三部分：UGC（User Generated Content，用户原创内容）、PGC（Professionally-generated Content，专业生产内容）和 OGC（Occupationally-generated Content，职业生产内容）。

PGC、OGC 和 UGC 三者之间既联系密切又有区别，相同之处在于它们都是用户生成的内容，区别在于 UGC 是（任意）用户所产生的内容，内容多元化，形式各异，质量参差不齐；PGC 在作为用户生产内容的同时也提供专业领域的优质内容，它们提供的内容偏专业化，在话术与专业领域上相比 UGC 来说，都更显优质，有利于传播；而 OGC 是职业的，他们在作为用户产生内容的同时也以提供内容为职业领取报酬。

3．用户画像体系组成

用户画像设计的标签体系有很多维度，根据不同标准，可以有不同类别，其中，**百度数据开放平台**对用户画像介绍中，从标签涉及内容角度，将标签体系分为 4 大类：生活日常、工作学习、兴趣爱好和消费倾向，具体如图 9.6 所示。

图 9.6 百度数据平台用户画像标签分类

依据用户属性变化频繁程度，一将用户数据划分为静态信息数据、动态信息数据两大类：

- 静态信息数据：相对短时间内变化不大的信息，主要指地域、年龄、性别、文化、职业、收入、生活习惯、消费习惯等人口属性和商业属性等；
- 动态信息数据：用户不断变化的行为信息，主要指产品类别、活跃频率、

产品喜好、产品驱动、使用习惯、产品消费等产品行为，在互联网上，用户行为，可以看做用户动态信息的唯一数据来源。

考虑到时间序列因素，标签体系又可以分为长期兴趣标签和短期兴趣标签、当前热门兴趣标签和突发事件兴趣标签等。

4．用户画像存储

用户画像是整个推荐系统中的核心数据，查找和更新很频繁，为了便于线上业务操作，一般我们存储选择如下：

- **长期兴趣标签和常态标签**：一般更新频率为小时、天或周，但查询频率比较高，内容比较多，因此这里建议采用 WT 存储引擎的 MongoDB3.0。
- **短期兴趣标签**：一般针对当前活跃用户更新频率比较高，当前行业都是分钟级或秒级，根据本企业数据规模，建议采用 Redis3.X，一般都会设置 TTL，自动清除过期数据。
- 对于用户画像来源——**原始用户行为数据**，由于内容比较多，读少写多，这里建议采用 HBase1.X，为提高读写性能，根据数据规模，初始设置合理分区个数。

9.1.5 元数据索引库

对任何推荐系统而言，元数据索引库都是最基础也是最核心的。这里所述的元数据是指业务产品基本信息。像电商商品、视频网站视频、媒体门户新闻，以及横向跨越产品界线的广告等，都可以称作元数据。因为这些数据的目的都是展示给最终用户，所以其索引设计好坏与存储直接关系到系统性能。

因为元数据一般都包括很多维度字段信息，而且一般其获取方式也是多种多样的，且更新频繁，所以我们需要考虑一种索引库满足读写频繁的基本需求，且能方便多维度查找，这样，考虑到目前现行的数据库引擎，只有全文检索满足需要，第 7 章专门给大家介绍过 ES2.x，正好符合当前元数据索引库的需要。

ES 索引库本身可以看成一个比较特殊的 NoSQL，其易用性、高效性、易扩展性、方便部署维护 HA 等均为当前元数据索引库的最佳选择。ES 在业界使用非常广泛，本身很有活力，发展非常迅速，是索引库的不二之选。

9.1.6 用户推荐服务

用户推荐服务模块是整个推荐系统中直接面向终端用户的，相当于推荐系统的前端，这里，可以基于各自企业实际需要，提供 Web Service 或 RPC 服务 API。在第 8 章专门给读者朋友介绍了当前微服务架构的相关技术，这里，作者建议：

- 如果服务使用 Web Service，可以考虑使用轻量级的嵌入式 Jetty 提供相关服务 API；
- 如果服务使用 RPC，可以考虑使用 Facebook 内部广泛使用的 Nifty。

当然这里作者只是建议给出了服务 API 的实现方式，具体业务需求和相关排序算法策略等就需要根据各自需要，自行把控了。

9.2 新闻推荐中用户画像近实时更新设计

上节主要从整体上介绍个性化推荐系统架构，这也是作者本人多年从事 NLP、数据挖掘和个性化推荐工作的经验总结，虽然作者有过多次独立完成整个大中型推荐系统的设计开发经历，但限于本书内容范围，这里重点向读者介绍新闻个性化推荐中用户画像实时更新模块的设计。

当前新闻推荐中，不管是新用户还是老用户，其兴趣点都是多种多样，而且随时间变化，往往这一时刻喜欢体育类新闻，下一时刻就喜欢明星类，或者下一时刻开始喜欢养生类，为了满足用户兴趣的多样性需求，这就要求我们推荐新闻给用户时：

- 紧紧抓住当前用户的兴趣点所在，为之推荐当前时刻最感兴趣的新闻；
- 同时还要给用户推荐满足其多样性口味的新闻。

只有满足这两点，才可以让用户喜欢上你的推荐引擎，从而喜欢你的产品。抓住用户的兴趣口味，才能抓住用户。用户画像近实时更新的意义不仅在于终端展示更加吸引用户，还可以提高新闻等媒体的个性化 push 乃至广告及电商 CTR，增强精准营销效果。

图 9.7 给我们描述了一个完整的用户画像近实时更新流程，这里以新闻推荐为例向大家介绍。完整的用户画像近实时更新主要包括两部分：终端用户推荐服务和用户画像近实时计算，前者主要是辅助后者，为其提供计算所需的一些基本信息，虽然它本身也是非常重要的一部分，但本书重点在于后者。这里我们假定，用户长期兴趣标签更新部分已经完善，此处只是简单介绍，这里要做的是在此基础上，近实时更新用户短期兴趣标签。

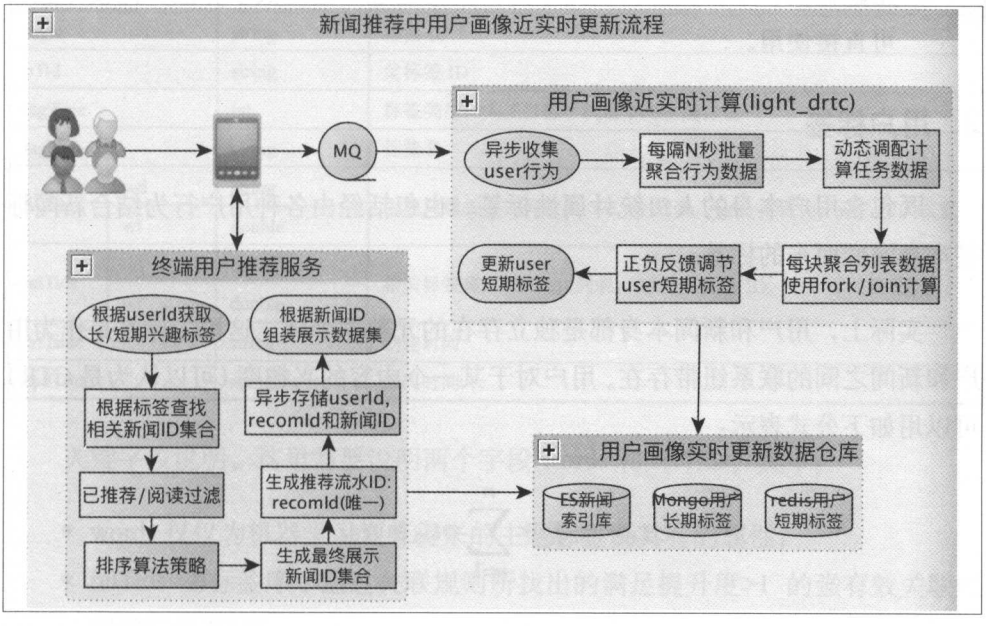


图 9.7 用户画像近实时更新流程

所谓近实时更新而不是实时更新，是因为实际情况，就算用户行为异步收集后立即做相关计算并更新，也会有一定延时，我们不可能完全做到实时更新，当前行业用户画像一般做到 5 秒至 6 秒更新就已经是比较优秀的计算平台。

9.2.1 新闻推荐中用户画像构成

新闻推荐中用户画像兴趣标签库主要包含两部分：新闻标签和用户标签。

1．新闻标签

既包含小编及用户给新闻显示打的标签，也包含对海量用户浏览过的新闻（一般是千万级或亿级）经过主题模型等机器学习手段得到的数十万、甚至百万级主题标签。

- 小编标签相对专业，一般可以直接使用。
- 用户标签，参差不齐，一般需要经过统计过滤掉一些噪声信息，才可使用。
- 主题标签，本身就源自新闻本身，而且经过严格的数学算法，比较专业，可直接使用。

2．用户标签

既包含用户本身的人口统计属性标签，也包括经由各种用户行为结合新闻标签本身演变而来的标签。

实际上，用户和新闻本身都是独立存在的元数据个体，这里，标签是作为用户和新闻之间的联系纽带存在。用户对于某一个内容的兴趣度（可以认为是 CTR）可以用如下公式表示：

$$r_{uc} = q_c * \sum_{i=1}^{n} m_{ci} * n_{ui}$$

其中 i=1，…，N 是内容具有的标签，$m(ci)$指的内容 c 和标签 i 的相关权重，$n(ui)$指的是用户 u 的标签 i 的权重值，q_c 指的是内容 c 的质量，可使用点击率表示。

系统中，一般针对老用户，由于其之前累积了比较多的用户行为，可以很容易知道其兴趣所在，而对于新用户，一般在新用户注册时，会让用户自行选择一些感兴趣的标签，同时结合海量用户行为统计得来的热门兴趣标签为新用户推荐

相关新闻。

9.2.2 新闻推荐中用户画像标签数据字典

如图 9.7 所示的整个用户画像实时更新流程，从数据库角度讲，主要涉及标签表、用户兴趣标签表、新闻标签表和用户推荐历史表，下面为其详细说明。

1. 标签tags数据字典

标签表 tags 结构如表 9.1 所示。

表 9.1 标签表 tags 结构

字 段		数据类型	说 明
tagId		string	标签 ID
pTid		string	父标签 ID
tagType		int	标签类型：人工打标、机器学习主题标签
tagName		string	标签名
words	wd	string	主题向量关键词列表，<wd,wt>列表，wd：关键词，wt：相关权重
	wt	double	
relTids	tid	String	相关标签集，<tid, wt>列表，tid:相关标签 ID，wt：相关权重
	wt	double	
crtTime		string	创建时间
updTime		string	修改时间

关键字段说明，这里需要说明两个字段 words 和 relTids，其中：

- words 仅仅为机器学习训练得来的主题标签所具有的属性；
- relTids 为标签库中通过关联规则所找出的满足提升度>1 的强有效关联规则的标签 ID 集。

标签库中，不管人工标签还是机器学习标签，都不会更新得很频繁，因为标签生成本身需要海量数据的计算，尤其是机器学习得来的标签。这里给大家简单说下**主题标签生成步骤**：

(1) 对海量（千万级或亿级）新闻文本，利用中文分词器 IK 或结巴等对其切

分词，并统计关键词基本的 tf、idf。

（2）利用 Word2vector 得到上述关键词的相似度模型，找出近义词列表向量，并作用到每个文本的词向量，这样就会使得每个文本的词向量得到极大扩展。

（3）上述基础上，利用凝聚层次聚类算法，对文本向量自底向上，计算相似度，并将相似度大于预设阈值的两个向量合并，依次递归，直到最终簇向量达到预设个数，即形成最终的主题向量。当然这中间，簇向量会过滤掉一些关键词，以降低簇向量纬度。为减小层级聚类算法的总体计算量，可以事先利用 Jaccard 相似度（交集大小与并集大小的比例）过滤掉相似度低于最小阈值而不需要计算的两两文本对。

标签库 tags 平时查询比较多比较适合放在 MongoDB 中存储。

2．用户兴趣标签数据字典

用户兴趣标签主要包含长期兴趣标签 longTermUtag 表（结构如表 9.2 所示）、用户长期兴趣表标签跨度表 userTagSpan 表、短期兴趣标签 shortTermUtag 表。

（1）longTermUtag 表结构

表 9.2　longTermUtag 表结构

字　段		数据类型	说　明
deviceId		string	设备 ID
userId		string	注册用户 ID
relTids	tid	string	直接相关标签集，值为<tid, wt, updTime>列表，tid:相关标签 ID，wt:相关权重，updTime：标签修改时间
	wt,	double	
	updTime	string	
expUTids	tid	string	相关用户扩展标签集，值为<tid, wt>列表，tid:相关标签 ID，wt:相关权重,一般取 top N
	wt	double	
expTTids	tid	string	相关标签扩展标签集，值为<tid, wt>列表，tid:相关标签 ID，wt:相关权重,一般取 top N
	wt	double	

（2）对于 longTermUtag 关键字段，这里需要说明一下 relTids、expUTids 和 expTTids 三个字段，其中：

- relTids 一般为用户的各种行为所作用的资源所属标签及其权重，需要结合一定的算法模型计算而来。这里给出一个简单有效的标签权重计算方法：

  1）用户原始行为日志中找出当前用户所作用过的所有新闻 ID 及其所属标签 tagId 集；

  2）统计出当前用户所浏览过的所有新闻 ID 个数及每个标签下的新闻 ID 个数；

  3）relTids 中每个标签权重初始即为标签下新闻 ID 个数除以该用户浏览的新闻总数。

  后期的周期更新都是在之前基础上累加的，但需要考虑时间衰减因素。因为用户的兴趣是随时间衰减的，即离当前时间越远的兴趣比重越低。对于时间衰减函数，我们可以参考“**牛顿冷却定律**”得到如下公式：

  本期权重 = 上期权重 * exp(-(冷却系数) * 间隔时间周期

  这里，“间隔时间周期”可以为天数或小时数，“冷却系数”一般都是根据实际数据经过 MATLAB 拟合而来的。

- expUTids 一般通过基于 user 的协同过滤找出相关用户的相关标签集，其计算步骤如下：

  1）通过协同过滤找出最相关的 top *K* 个相关用户及相关权重；

  2）取出这 K 个用户“relTids”，并对其中标签权重 wt 均乘以当前用户相关权重。

  3）将上述所有标签按照其新的相关度高低排序，取出与 relTids 中不重复的 top N 个 tagId 作为最终结果。

- expTTids 获取步骤如下：

  1）取出 relTids 所有标签及权重。

2）从标签表 tags 中找出每个标签的扩展标签集，并对扩展标签集的每个标签权重均乘以当前标签权重得到最终扩展标签权重。

3）将上述所有标签按照其新的相关度高低排序，取出与 relTids 及 expUTids 中不重复的 top*N* 个 tagId 作为最终结果。

从上述标签权重计算方法，可以了解到用户长期兴趣标签更新比较复杂，尤其是面对海量用户行为数据，因此长期兴趣标签更新周期相对较长，一般时间周期为天或数小时，而且更新在上次计算基础上只针对当前时间段内的活跃用户。字段 expUTids 和 expTTids 中的取值个数 top *N*，是指各自综合相关度最高的 top*N* 个，其中 *N* 一般取值不会太大，可以根据实际情况，这里暂定 20，而且这里的 tid 均为 relTids、expUTids 和 expTTids 是不重复的。

（3）用户长期兴趣表标签跨度表 userTagSpan

长期兴趣标签随着用户的使用而长期积累，积累到一定阶段，单个用户的长期兴趣标签内容可能会比较大，标签很多，多则达上千甚至上万，但实际使用时，我们不可能用这么多，一般我们会选择使用该用户近期感兴趣的 top *N* 个标签。

longTermUTag 这里存储在 Mongo 中，而 MongoDB3.0 中有单条数据不超过 16MB 限制，因此如果某个用户的 longTermUtag 内容达到临界点，这时需要把该用户的长期兴趣标签依据时间序列从新到旧排序，将 longTermUtag 拆分存在相应散列表 longTermUtag_N 中，其中 *N* 从 0,1,2 开始逐渐增大。其中 longTermUtag 为主表，而且存储的是都是用户最新参与的兴趣标签及权重。因此需要专门设置一个用户长期兴趣表标签跨度表 userTagSpan 用于存储用户长期兴趣标签过大而散列在多个表的用户记录。表 userTagSpan 结构如表 9.3 所示。

（4）表 shortTermUtag

其字段为表 longTermUtag 子集，仅包含 **deviceId**，**relTids** 两个字段。shortTermUtag 这里预设存储在 Redis 中，而且在将用户短期实时标签存入 Redis 中时，即设置 TTL，其 TTL 时间周期一般小于长期兴趣标签更新周期。这里短期兴趣标签之所以不包含 userId、expUTids 和 expTTids 字段，目的是尽量减少 Redis

缓存和降低 onlinelearning 的计算复杂度，提高效率。

表 9.3 表 userTagSpan 结构

字 段	数据类型	说 明
deviceId	string	设备 ID
userId	string	用户 ID
tables	List<string>	用户长期兴趣标签拆分后横跨的表名列表
updTime	String	修改时间
crtTime	string	创建时间

因为短期兴趣标签计算一般都是近实时更新的，一般更新周期为分钟及秒级，这里我们暂定更新频率为 10 秒，也就是从用户行为发生到用户短期兴趣标签更新完成的时间，其详细更新流程下面会有详细说明。

3. 新闻标签news_tag数据字典

（1）表 news_tag 结构

表 news_tag 结构如表 9.4 所示。

表 9.4 表 news_tag 结构

字 段		数据类型	说 明
newsId		string	新闻 ID
title		string	标题
author		string	作者
digest		string	摘要
keywords		string	关键词
content		string	内容
pubTime		string	发布时间
visitUrl		string	静态访问 URL
relTids		list<string>	用于检索的标签 ID 列表
jsTids	tid	string	用于计算的<tid,wt>标签权重元数据列表，tid:相关标签 ID，wt:相关权重
	wt	double	
updTime		string	修改时间

（2）news_tag 关键字段说明

这里仅需要说明两个字段：relTids 和 jsTids，其中：

- relTids 用于根据标签 ID 检索相关新闻，考虑到 ES 索引特点，relTids 仅用于索引，可以不存储，该字段用于面向终端用户推荐服务时根据用户相关标签获取相关新闻；
- jsTids 字段信息为<tid,wt>元数据对列表，用于用户画像实时更新和新闻推荐服务，根据 newsId 获取所属标签及权重。

（3）新闻实时更新

新闻标签索引表，不仅经常用于根据 newsId 获取新闻，还可以用于根据标签 ID 检索返回相关新闻，有时还考虑到时间维度、关键词检索维度等，因此将 news_tag 存储在 ES 是比较合适的，而且 ES 天生支持实时更新。

这里我们借助当前炙手可热的 Kafka 作为新闻实时更新所需的 MQ，只要业务方有新的新闻变更，立即将变更信息发送到 Kafka，同时监听 Kafka 中相关主题的索引更新服务在接收到数据后就立即更新到 ES 索引库，从而完成新闻索引库的异步实时更新。

4．用户推荐历史表histRecom数据字典

histRecom 表结构如表 9.5 所示。

表 9.5　histRecom 表结构

字　段	数据类型	说　明
recomId	string	推荐流水 ID
deviceId	string	设备 ID
nids	List<string>	推荐的新闻 ID 集
tids	List<string>	推荐新闻所属标签 ID 集
crtTime	string	流水产生时间

用户推荐历史主要用于用户画像近实时更新的隐性的正负反馈计算中，终端用户的每次点击浏览、收藏或分享的新闻 ID 均有一个相应的推荐流水 recomId，根据 recomId 迅速定位该新闻 ID 来自哪次推荐流水，从而针对该次推荐就可以知

道推荐的新闻 ID 集中用户显示感兴趣的和不感兴趣的标签个数比例，即正负反馈，从而根据相应数学模型实时更新当前用户的短期兴趣标签。

由于用户的推荐历史流水数据主要用于短期兴趣标签标签的实时更新，计算后就无太大作用，为减少数据库库存压力，因此在 MongoDB 中事先创建好 TTL 索引，超时周期这里暂定 2 天，超过当前时刻 2 天的流水数据，MongoDB 会自动删除。

用户推荐流水中的 nids 还可以根据用户终端行为，近实时统计每个新闻 ID 的 CTR。这里只需要一个新闻 CTR 计算服务：每隔 1 分钟统计一下最新的每个新闻 ID 的曝光次数，以及所接受的分布式实时计算服务中数据收集节点所传送来的新闻点击次数，即可得到当前时刻的活跃新闻 CTR 值。

9.2.3 新闻推荐用户画像实时更新流程

如图 9.7 所示，用户画像近实时更新需要涉及两部分：面向终端用户的推荐服务和用户画像实时计算服务，其中用户画像实时计算我们计划用本书作者所开源的 light_drtc，下面详细介绍其更新流程。

（1）面向终端用户的推荐服务中，伴随着给用户的每次推荐新闻 ID 集，即 newsIdList，由系统产生一个唯一推荐 ID，即 recomId，这时我们可以将数据：recomId，deviceId，newsIdList 及所对应的标签 tag 集和当前时刻 curTime，以 recomId 为主键“_id”，使用 MongoDB3 的异步驱动，异步插入到 MongoDB3.x 的“histRecom”表中。

这里 recomId 的生成，我们可以考虑使用 UUID（Universally Unique Identifier）全局唯一标识符，JDK1.5 中新增的一个类，在 java.util 下，用它可以产生一个号称全球唯一的 ID。因为 UUID 是指在一台机器上生成的数字，它保证对在同一时空中的所有机器都是唯一的。按照开放软件基金会（OSF）制定的标准计算，用到了以太网卡地址、纳秒级时间、芯片 ID 码和许多可能的数字。

（2）将包含 recomId 的推荐结果推送到展示终端，当用户点击、收藏或分享

推荐结果时，此时，我们可以记录下：deviceId，recomId，newsId 和行为类型，并发送到 MQ，这里为了便于后续工作开展，我们选择 Kafka，routingKey 请选择使用 deviceId。

（3）CN（数据收集服务节点）从 MQ 实时获取数据后，对每条数据的唯一标识 ID，使用 Guava 的 MurmurHash 算法利用 AN 个数计算 routingKey 哈希值，将该条数据暂存于 CN 中的相应 AN（**计算任务管理节点**）ID 对应的本地队列中，然后每隔 2 秒执行一个多线程任务（线程个数依赖于 AN 个数）：从当前本地队列中取出数据列表，批量推送给 AN。这样可以保证每个用户的行为信息均会定向推送到固定 AN。

这里对所接受的每个新闻 ID，均放置在 CN 的一个专门用于存放 newsId 的本地队列中，随着前述间隔 2 秒的任务周期，统计出最近 2 秒内每个新闻的 click 次数，并将该数据发送到新闻 CTR 统计服务中。

（4）AN 将收到的用户行为数据暂存于本地队列中，AN 每隔 6 秒钟将当前本地队列中的用户行为以 userId 为基准进行数据聚合，加工成每条数据信息形如：

```
{
    "uid":"设备 ID 或用户 ID",
    "data": {
        "click": {$docId : $recomId},
        "collect": {$docId : $recomId},
        "share": {$docId : $recomId}
    }
}
```

的信息列表 userActionList，将 userActionList 根据当前 AN 所管辖的 JN（任务计算节点）个数 jnNum，均分后推送给各个 JN，并监控每个 JN 计算任务状态，如果失败，重新计算相应任务。当然这其中，如果当前任务开始计算时，上次计算任务还没有结束，则需要参考 light_drtc 中的策略，结合内存和本地硬盘将数据暂存本次任务数据，这样可以保证即使某次计算任务比较大，也不会导致内存开销过大，从而影响计算服务的稳定性。

（5）JN 对接受的来自 AN 的聚合用户行为数据 usearActionList，依据其用户个数 userNum，设立 userNum 个计算任务，每个计算任务独立计算并更新每个用

户的短期兴趣标签权重，这里每个计算任务都是独立的，相互没有任何依赖，为此，正好可以借助 JDK7 起具备的 fork/join 并行计算框架，对于单个计算比较复杂的流程，我们使用 JDK8 的新特性 CompletableFuture 完成异步执行，提高效率。

为了加快整体计算效率，降低 Mongo 和 Redis 及 ES 访问频繁度，对于 JN 中 fork / join 下的每个任务，可以对该任务下所有用户浏览的新闻 ID 集一次性从 ES 中批量查询出每个新闻所属标签 ID 及权重，一次性从 Mongo 中获取所有 recomId 集对应的推荐流水、一次性获取所有 deviceId 所对应长期及短期兴趣标签。

(6) JN 中每个计算任务都利用正负反馈更新其相应权重：对于用户有关浏览、收藏或分享行为中的每个新闻 ID，根据其对应的$recomId，均可以快速定位到该新闻来自哪次推荐，在该次推荐的 newsId 集中，当前行为所作用的 newsId 所属的标签即为用户当前时刻比较感兴趣的，即为正反馈；该次推荐历史中其他未被点击的 newsId 所属标签集则属于用户相对不太感兴趣的，即为负反馈。一般而言，正反馈标签提高权重，负反馈标签降低权重。

(7) 上述利用正负反馈对用户短期兴趣标签进行权重升降，本身就相当于前节所述的利用逻辑回归 LR 进行标签权重预估，因此，这里我们建议大家参考：

$$\sigma(X) = 1/(1+\exp(m*X))$$

考虑到推荐服务中对用户长、短期兴趣标签合并问题，这里在计算用户短期兴趣标签权重时，考虑到时间序列，长期兴趣标签会随着时间衰减，因此我们可以考虑采用：

$$curShortTagWeight = \sigma * \text{shortTagWeight} + (1 - \sigma) * \text{longTagWeight}$$

这样在用户推荐服务时，如果有短期兴趣标签，直接使用短期标签，否则使用长期标签（每天用户首次访问使用），减少前端不必要计算，提高前端效率。

当然这里的 LR 参数 m 和时间序列参数 σ 都是根据系统中真实数据，利用 MATLAB 等拟合而来的，这样才会让公式更加适合当前业务需要。

9.3 新闻推荐用户画像近实时更新技术实现

用户画像实时更新，需要流式分布式实时计算，当前行业用得比较广的分布式实时计算框架是 Storm、SparkStreaming，这里会分别介绍二者如何用到本项目中，再综合对比作者本人研发的 light_drtc。

用户画像的分布式实时计算，主要包括三部分：实时接受用户行为数据流、流数据加工转换、短期兴趣标签权重更新。实时接受用户行为数据流，一般都是通过接入 MQ，源源不断地接收用户行为数据，三个框架都分别提供接入 MQ 的接口；流数据加工转换三个框架也都由各自不同方式来完成；最后的短期兴趣标签权重更新则是业务逻辑及算法模型的具体实现，这点三者差别不大。下面以三个框架计算实例介绍。这里 MQ 我们选用 Kafka。

9.3.1 Storm 接入 Kafka 实时计算实例

本实例，主要向读者介绍 Storm 接入 Kafka 完成对所接受的信息流中单词计数的实时统计，开发前，我们先引入相关 Maven 依赖：

```
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>0.9.5</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-kafka</artifactId>
    <version>0.9.5</version>
</dependency>
```

下面是完整实例代码，并有相关详细注释。

```
import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.atomic.AtomicInteger;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.generated.AlreadyAliveException;
import backtype.storm.generated.InvalidTopologyException;
import backtype.storm.spout.SchemeAsMultiScheme;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import storm.kafka.BrokerHosts;
import storm.kafka.KafkaSpout;
import storm.kafka.SpoutConfig;
import storm.kafka.StringScheme;
import storm.kafka.ZkHosts;

public class HiStromKafka {
    //显示所接受的 tuple 时间
    private static SimpleDateFormat df = new SimpleDateFormat
("yyyyMMddHHmmss");

    //对所接受的日志信息的初步处理：空格分词，并负值 1
    public static class KafkaWordSplitter extends BaseRichBolt {
        private static final long serialVersionUID = 1L;
        private OutputCollector collector;

        @Override   //Bolt 初始化
        public void prepare(Map stormConf, TopologyContext context,
                            OutputCollector collector) {
            this.collector = collector;
        }

        @Override   //信息处理，每个 Tuple 单独处理
        public void execute(Tuple input) {
            System.out.println(df.format(new Date())+"\t"+input);
            //默认每个 tuple 只有 1 个字段信息
            String line = input.getString(0);
            String[] words = line.split("\\s+");//空格分词
            for(String word : words) {
                System.out.println(df.format(new Date())
                        +"\tEMIT[splitter -> counter] \t"+input);
                //传送到下一个 Bolt
                collector.emit(input, new Values(word, 1));
            }
            collector.ack(input);//向 Kafka Spout 确认受到的 tuple
```

```
    }

    @Override
    public void declareOutputFields(
                    OutputFieldsDeclarer declarer) {
        //下游bolt 中tuple含有2字段
        declarer.declare(new Fields("word", "count"));
    }
}

//第二个Bolt:完成第一个bolt对kafka 消息空格分词后的单词个数统计
public static class WordCounter extends BaseRichBolt {
    private static final long serialVersionUID = 1L;
    private final Log LOG
                    = LogFactory.getLog (WordCounter.class);
    private OutputCollector collector;
    private Map<String, AtomicInteger>counterMap;

    @Override//Bolt初始化
    public void prepare(Map stormConf, TopologyContext context,
                                OutputCollector collector) {
        this.collector = collector;
        //单词计数
        this.counterMap = new HashMap<String, AtomicInteger>();
    }

    @Override
    public void execute(Tuple input) {
        String word = input.getString(0);
        int count = input.getInteger(1);
        System.out.println(df.format(new Date())
                +"\tRECV[splitter -> counter]  \t"
                +word + " : " + count);
        AtomicInteger ai = this.counterMap.get(word);
        if(ai == null) {
            ai = new AtomicInteger();
            this.counterMap.put(word, ai);
        }
        ai.addAndGet(count);//完成单词计数累加
        collector.ack(input);
        System.out.println(df.format(new Date())
                +"\tCHECK statistics map: \t"
                + this.counterMap);
    }

    @Override//清除掉过期或无效信息
    public void cleanup() {
        LOG.info("The final result:");
        Iterator<Entry<String, AtomicInteger>> iter
                    = this.counterMap.entrySet().iterator();
        while(iter.hasNext()) {
            Entry<String, AtomicInteger> entry = iter.next();
```

```
                LOG.info(entry.getKey() + "\t:\t"
                        + entry.getValue().get());
            }
        }

        @Override
        public    void    declareOutputFields(OutputFieldsDeclarer
declarer) {
            //此处不需要声明，因为不再需要向下传送数据
        }

    }

    public static void main(String[] args) {
        //kafka 所依赖的 zookeeper 集群地址，根据实际情况调整
        String kafkaZkHosts = "192.168.10.11:2181,192.168.10.12:
2181,192.168.10.13:2181";
        //Kafka 消息主题
        String kafkaTopic = "storm_kafka_test";
        //kafka 所依赖 zookeeper 路径
        String kafkaZkRoot = "/storm_kafka_test";
        //stormClientId 用于标识每个 Storm 拓扑
        String stormClientId = "light2016";

        BrokerHostsbrokerHosts = new ZkHosts(kafkaZkHosts);
        SpoutConfigspoutConf = new SpoutConfig(brokerHosts,
                kafkaTopic, kafkaZkRoot, stormClientId);
        spoutConf.scheme = new SchemeAsMultiScheme(
                                        new StringScheme());
        spoutConf.forceFromStart = false;
        //此处 zookeeper 集群地址为 storm 所依赖集群，根据实际情况调整
        spoutConf.zkServers = Arrays.asList(new String[]{
            "192.168. 20.21", "192.168.20.22","192.168.20.23"});
        spoutConf.zkPort = 2181;//Storm 所依赖 zookeeper host 端口

        TopologyBuilder builder = new TopologyBuilder();
        ////这里 spout 并行度一般设置为与 Kafka 分区个数保持一致
        builder.setSpout("kafka-reader",
                        new KafkaSpout(spoutConf), 3);

        //shuffleGrouping 将流分组定义为混排。
        //这种混排分组意味着来自 Spout 的
        //输入将混排，或随机分发给此 Bolt 中的任务。
        //shuffle grouping 对各个 task 的 tuple 分配得比较均匀。
        builder.setBolt("word-splitter", new KafkaWordSplitter())
                        .shuffleGrouping("kafka-reader");

        //fieldsGrouping 机制保证相同 field 值的 tuple 会去同一个 task，
        //这对于 WordCount 来说非常关键，如果同一个单词不去同一个 task，
        //那么统计出来的单词次数就不对了。
        builder.setBolt("word-counter", new WordCounter())
```

```
                .fieldsGrouping("word-splitter", new Fields("word"));

        Config conf = new Config();
        String name = HiStromKafka.class.getSimpleName();
        if (args != null && args.length > 0) {//远程平台执行方式
            // Nimbus host name passed from command line
            conf.put(Config.NIMBUS_HOST, args[0]);
            conf.setNumWorkers(3);//并行
            try {
                StormSubmitter.submitTopologyWithProgressBar (
                    name, conf, builder.createTopology());
            } catch (AlreadyAliveException e) {
                e.printStackTrace();
            } catch (InvalidTopologyException e) {
                e.printStackTrace();
            }
        }else{//本地模式执行
            conf.setMaxTaskParallelism(3);//3 个 executor 并行
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology(name, conf,
                                builder. createTopology());
            try {
                Thread.sleep(60000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            cluster.shutdown();
        }
    }
}
```

上述实例，给读者朋友演示了使用 Storm 接入 Kafka，逐条获得信息流数据 Tuple，经过单词拆分和单词计数 2 个 bolt 处理过程，最终将统计结果打印出来。将项目打包后，利用如下命令：

storm jar HiStromKafka.jar org.light.lrtdc.job.HiStormKafka

将任务提交到 Storm 集群，即可在 Storm 控制台查看其运行状况。当然开发阶段也可以以本地模式执行，方便调试开发。

Storm 不仅可以逐条实时处理信息，也可以小批量地一次处理几条信息，当然这里我们要参考 Storm 的事务处理 trident 方式，其使用方式和上面实例类似，不同之处在于 trident 每次处理一组信息，相当于微批处理。

9.3.2 Spark Streaming 接入 Kafka 实时计算实例

1. Spark Streaming接入Kafka的两种方式

Spark Streaming 从 Kafka 中接收数据，这里将会介绍两种方法[14]：使用 Receivers 和 Kafka 高层次的 API；使用 Direct API，这是使用低层次的 KafkaAPI，并没有用到 Receivers，是 Spark 1.3.0 中开始引入的。这两种方法有不同的编程模型、性能特点和语义担保。

（1）基于 Receivers 的方法

这个方法使用了 Receivers 来接收数据。Receivers 的实现用到 Kafka 高层次的消费者 API。对于所有的 Receivers，接收到的数据将会保存在 Spark executors 中，然后由 Spark Streaming 启动的 Job 来处理这些数据。

在创建 DStream 的时候，你也可以指定数据的 Key 和 Value 类型，并指定相应的解码类。需要注意的是：

- Kafka 中 Topic 的分区和 Spark Streaming 生成的 RDD 中分区不是一个概念。所以，在 KafkaUtils.createStream()增加特定主题分区数仅仅是增加一个 receiver 中消费 Topic 的线程数，并不增加 Spark 并行处理数据的数量；
- 对于不同的 Group 和 topic，我们可以使用多个 receivers 创建不同的 DStreams 来并行接收数据。

然而，在默认配置下，这种方法在失败的情况下会丢失数据，为了保证零数据丢失，你可以在 Spark Streaming 中使用 WAL 日志，这是在 Spark 1.2.0 才引入的功能，这使得我们可以将接收到的数据保存到 WAL 中（WAL 日志可以存储在 HDFS 上），所以在失败的时候，我们可以从 WAL 中恢复，而不至于丢失数据。

（2）基于 Direct API 的方法

---

14 Spark Streaming 和 Kafka 整合 http://dataunion.org/15193.html

和基于Receiver接收的数据不一样，这种方式定期地从Kafka的topic+partition中查询最新的偏移量，再根据定义的偏移量范围在每个 batch 里面处理数据。当作业需要处理的数据来临时，spark 通过调用 Kafka 的简单消费者 API 读取一定范围的数据。

和基于 Receiver 方式相比，这种方式主要有以下几个优点：

- 简化并行。我们不需要创建多个 Kafka 输入流，然后 union 它们。而使用 directStream，Spark Streaming 将会创建和 Kafka 分区一样的 RDD 分区个数，而且会从 Kafka 并行地读取数据，也就是说，Spark 分区将会和 Kafka 分区有一一对应的关系。
- 高效。第一种实现零数据丢失是通过将数据预先保存在 WAL 中，这将会复制一遍数据，这种方式实际上很不高效，因为这导致了数据被拷贝两次：一次是被 Kafka 复制；另一次是写到 WAL 中。但是这里介绍的方法因为没有 Receiver，从而消除了这个问题，所以不需要 WAL 日志。
- 恰好一次语义（Exactly-once semantics）。通过使用 Kafka 高层次的 API 把偏移量写入 Zookeeper 中，这是读取 Kafka 中数据的传统方法。虽然这种方法可以保证零数据丢失，但是还是存在一些情况导致数据会丢失，因为在失败情况下通过 Spark Streaming 读取偏移量和 Zookeeper 中存储的偏移量可能不一致。本章提到的方法是通过 Kafka 低层次的 API，并没有用到 Zookeeper，偏移量仅仅被 Spark Streaming 保存在 Checkpoint 中。这就消除了 Spark Streaming 和 Zookeeper 中偏移量的不一致，而且可以保证每个记录仅仅被 Spark Streaming 读取一次，即使是出现故障。

此方法唯一的坏处就是没有更新 Zookeeper 中的偏移量，所以基于 Zookeeper 的 Kafka 监控工具将会无法显示消费的状况。然而你可以通过 Spark 提供的 API 手动地将偏移量写入到 Zookeeper 中。

2．完整实例介绍

下面是类HelloSparkStreaming.java为SparkStreaming的Minid-Batch实时统计信息流中单词计数的实例，其中包含两部分测试：第一个测试只是 Spark Streaming

接入普通的 TCP 网络流，借助 NC 实现；第二个实例借助 Kafka 实现，是借助上小节中第一种方式，Receivers 方式，开发前，引入 Spark 相关 Maven 依赖：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>1.6.2</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.11</artifactId>
    <version>1.6.2</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka_2.11</artifactId>
    <version>1.6.2</version>
</dependency>
```

下面是完整代码实例：

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Pattern;

import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;

import org.apache.spark.streaming.*;
import org.apache.spark.streaming.api.java.*;

import com.google.common.base.Optional;
import com.google.common.collect.Lists;

import org.apache.spark.streaming.kafka.*;

import scala.Tuple2;

public class HelloSparkStreaming {
    private static final Pattern SPACE = Pattern.compile(" |,|:");

    //此处测试借助 NC 完成
    public static void test1() {
        JavaStreamingContext jssc = new JavaStreamingContext (
                                "local[2]",
                                "JavaNetworkWordCount",
                                new Duration (10000));
```

```
            //使用 updateStateByKey()函数需要设置 checkpoint
            jssc.checkpoint(".");
            // 打开本地的端口 9999
            JavaReceiverInputDStream<String> lines =
                        jssc.socketText Stream("localhost", 9999);
            // 按行输入，以空格分隔
            JavaDStream<String> words = lines.flatMap(
                        line -> Arrays.asList (SPACE.split(line)));
            // 每个单词形成 pair，如 (word, 1)
            JavaPairDStream<String, Integer> pairs = words.mapToPair(
                              word -> new Tuple2<>(word, 1));
            // 统计并更新每个单词的历史出现次数
            JavaPairDStream<String, Integer> counts = pairs
                .updateStateByKey ((values, state) -> {
                    Integer newSum = state.or(0);
                    for (Integer i : values) {
                        newSum += i;
                    }
                    return Optional.of(newSum);
                });
            counts.print();
            jssc.start();
            jssc.awaitTermination();
            jssc.close();
        }

        //Spark Streaming 接入 Kafka，对实时数据流每隔 2 秒统计其中单词计数
    public static void testKafka() {
            JavaStreamingContext jssc =new JavaStreamingContext (
                                        "local[2]",
                                        "SparkStreingKafka",
                                         new Duration(1000));
            // 使用 updateStateByKey()函数需要设置 checkpoint
            jssc.checkpoint(".");
            //配置读取的 Kafka 中主题相应的分区个数配置
            Map<String, Integer>topicPartions = new HashMap<String,
Integer>();
            topicPartions.put("sparkStreamingKafkaTest", 9);
            //构造一个 Kafka 数据流，构造参数：steamingContext、
            //Kafka 依赖的 zookeeper 集群地址、消费群组、主题分区配置
            JavaPairReceiverInputDStream<String, String>kafkaStream =
                KafkaUtils.createStream(jssc,
        "192.168.10.11:2181,192.168.10.12:2181,192.168.10.13:2181",
                "test_spark_streaming_kafka", topicPartions);
            //初步信息转换
            JavaDStream<String> lines = kafkaStream.map(
                new Function<Tuple2<String, String>, String>() {
                    @Override
                    public String call(Tuple2<String, String>tuple2) {
                        return tuple2._2();
                    }
```

```
        });
        //每条信息经过正则表达式拆分成数组列表
        JavaDStream<String> words = lines.flatMap(
                new FlatMapFunction <String, String>() {
                @Override
                public Iterable<String> call(String x) {
                    return Lists.newArrayList(SPACE.split(x));
                }
            });
        //实现类似于 Hadoop 的 map/reduce，完整最终统计
        JavaPairDStream<String, Integer>wordCounts =
            words.mapToPair(
                newPairFunction<String, String, Integer>() {
                    @Override
                    publicTuple2<String, Integer> call(String s) {
                        return new Tuple2<>(s, 1);
                    }
                }).reduceByKey(
                    new Function2<Integer, Integer, Integer>() {
                    @Override
                    public Integer call(Integer i1, Integer i2) {
                        return i1 + i2;
                    }
                });
        wordCounts.print();

        jssc.start();
        jssc.awaitTermination();
        jssc.close();
    }

    public static void main(String[] args) {
        //  test1();
        testKafka();
    }
}
```

图 9.8 所示为单独运行测试函数 test1()测试结果截图，服务启动后，终端使用 linux 命令 nc，命令格式：

```
nc -lk 9999
```

Linux 命令启动后，可以模拟网络流，实时发送一些数据，Spark streaming 监控相应服务端口，则实时获取 nc 端输入的测试数据，而在内部每隔 2 秒统计一批数据的单词计数。

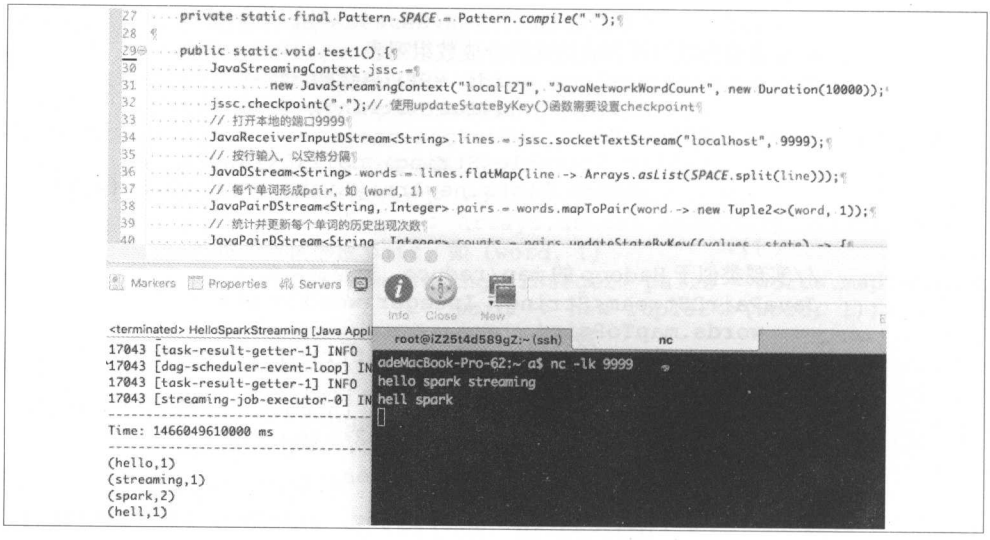


图 9.8 Spark Streaming 接入 NC 网络流统计单词

9.3.3 Light_drtc 接入 Kafka

作者在设计 light_drtc 时，考虑到当前行业 MQ 的使用热度，事先把当前最热门的两个 MQ：RabbitMq 和 Kafka 集成到了框架中，因此使用 light_drtc 非常方便，只需要在框架配置文件中配置好 Kafka 参数即可，相关参数如下：

```
#kafka config
# kafka 所依赖的 zookeeper 集群
kfZkServers=192.168.3.11:2181,192.168.3.12:2181,192.168.3.13:2181
# kafka 消费组 ID
kfGroupId=lrtdc_mq
kfAutoOffsetReset=smallest
# Kafka 消费主题
kfTopic=userAction
```

代码启用方式，请参考：

https://github.com/lrtdc/light_drtc/tree/master/src/test/java/org/light/ldrtc/test/AdminNodeServer.java。

9.3.4 用户画像实时更新核心实现

前述 3 节给大家演示了三种框架分别接入 Kafka 数据流的实例，框架一旦接

入 Kafka，我们只需要对所接受的数据加工，实现具体业务逻辑及相关算法模型即可，至于框架内部的任务调度，资源分配等都是框架本身已经考虑好的，所以我们只要专注业务逻辑和算法实现即可。这里限于篇幅，我们只给出系统视线中两个核心模块：MQ 数据聚合加工接口和标签实时更新算法模型。下面给出两部分实现核心模块。

1．用户行为聚合加工

如何使用 light_drtc，请参考第 3 章的使用说明。本节目的是将 AN 中本地对队列中的用户行为数据加工成计算任务易处理的方式，这里之所以将以设备 ID 为 key 聚合当下计算周期内用户的各种行为放置在 JSon 对象中，是为了计算的通用性。下面为完整代码。

```
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.ConcurrentLinkedQueue;
import org.light.rtc.base.StreamLogParser;
import com.alibaba.fastjson.JSONObject;

public class UserActionParser implements StreamLogParser {
    /**
     * 这里我们暂设队列中的每条数据信息如下：deviceId, actionType,
newsId, recomId
     * {
     *   "uid":"设备 ID 或用户 ID",
     *   "data": {
     *          "click": {$docId : $recomId},
     *          "collect": {$docId : $recomId},
     *          "share": {$docId : $recomId} }
     * }
     */
    @Override
    public    List<String>parseLogs(ConcurrentLinkedQueue<String>
dataQu, int curNum) {
        Map<String,Map<String,Map<String,String>>>rtMap
        = new HashMap<String,Map<String, Map<String,String>>>();
        String line = null;
        String[] fields = null;
        Map<String,Map<String,String>> actions = null;
        Map<String,String>docIds = null;
        for(int i=0; i<curNum; i++){
```

```
                line = dataQu.poll();//从数据队列中取值
                fields = line.split(",");
                //加工成聚合 Map 数据
                if(fields!=null && fields.length==4){
                    actions = rtMap.remove(fields[0]);
                    if(actions==null){
                        actions   =   new   HashMap<String,Map<String,
String>>();
                    }
                    docIds = actions.remove(fields[1]);
                    if(docIds==null){
                        docIds = new HashMap<String,String>();
                    }
                    docIds.put(fields[2],fields[3]);
                    actions.put(fields[1], docIds);
                    rtMap.put(fields[0],actions);
                }
            }
            //数据格式化为 Json
            List<String>rtList = new LinkedList<String>();
            JSONObjectrtJson = null;
            for(Entry<String,Map<String,Map<String,String>>> item
                                                    : rtMap.entrySet()){
                rtJson = new JSONObject();
                rtJson.put("uid", item.getKey());
                rtJson.put("data", item.getValue());
                rtList.add(rtJson.toJSONString());
            }
            return rtList;
        }

    }
```

在 ACN 启动前，需要将此实现类传递给 ACN 中的相应接口，以确保 ACN 正常启动服务，实时接受 MQ 数据。

2．用户短期兴趣标签实时更新

本节向读者演示短期兴趣标签实时更新的核心实现逻辑及相关算法模型，本节实现逻辑也是参考前述章节短期兴趣标签更新流程。程序处理流程如下：

- 继承 Fork/Join 框架，对接受的数据反序列化为结构数据。
- 为提高整体计算效率，减少对数据库访问压力，适当做了批量处理：
  - ➢ 批量获取用户短期兴趣标签及权重：shortUtags;。
  - ➢ 批量获取没有短期兴趣标签的用户的长期兴趣标签及权重：longUtags;。

- 整理出首次访问新用户：noTagUids。

- 对首次访问的新老用户的短期兴趣标签更新逻辑分成两个独立模块。
- 前述两个模块中，为加快整体计算效率，减少数据库访问压力，批量操作：
  - 批量查找系统为当下用户计算周期内的推荐历史中的新闻标签集 hisRecomTids;。
  - 批量查找当下用户计算周期内的所浏览分享等新闻标签集 curUserNtags;。
- 针对每个用户，根据其最新行为而隐含的正负反馈效应，计算最新标签权重。
  - 最新行为有关标签这里正负反馈，我们采用的是逻辑回归 LR 公式；
  - 用户之前老兴趣标签权重，因为时间衰减，我们参考“牛顿冷却定理”公式实现。
- 上述标签权重进行归一化，并批量存入 Redis。

这里主程序中引用的 RedisDao、MongoDao 和 EsDao，其中“cache”批量查找方法均采用 Google Guava Cache 的 CacheLoader 方式，使用本地缓存，以减少对数据库访问压力。下面为标签更新的完整主程序，其中有详细解释。

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.concurrent.RecursiveTask;

import org.light.rtc.dao.RedisDao;
import org.light.rtc.dao.EsDao;
import org.light.rtc.dao.MongoDao;
import org.light.rtc.util.Constants;

import com.alibaba.fastjson.JSONObject;
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;

public class ShortUtagStats extends RecursiveTask<Integer> {

    private static final long serialVersionUID = 807327367564
8538612L;
    private List<String>uActions;
```

```
//时间衰减比例因子，牛顿冷却因子小于此，以此为准
private final float redutionRatio = 0.7f;

public ShortUtagStats(List<String>joinedAction) {
    this.uActions = joinedAction;
}

@Override
protected Integer compute() {
    int rtNum = -1;
    if (this.uActions.size() <= Constants.minJobBatchNum) {
     Map<String,String>userLbs = new HashMap<String,String>();
        Map<String,Map<String,String>>tmpActions = null;
        JSONObjectrtJson = null;
      Map<String,Map<String,Map<String, String>>>joinActions
      = new HashMap<String,Map<String, Map<String,String>>>();
        //本次计算所有用户作用的 newsId 集
        Set<String>newsIdSet = new HashSet<String>();
        //本次计算所有用户作用的 newsId 来源 recomId 集
        Set<String>recomIdSet = new HashSet<String>();
        for(String rt : this.uActions){
            rtJson = JSONObject.parseObject(rt);
            tmpActions = (Map<String,Map<String,String>>)
                         rtJson.get("data");
            if(tmpActions!=null &&tmpActions.size()>0){
                joinActions.put(rtJson.getString("uid"),
                                                tmpActions);
                for(Map<String,String> nr
                            : tmpActions. values()){
                    newsIdSet.addAll(nr.keySet());
                    recomIdSet.addAll(nr.values());
                }
            }
        }
        //批量找出所有用户短期兴趣标签，
        //内容格式：<设备 ID，<标签 ID，相关权重>>
        Map<String,Map<String,Double>>shortUtags =
                        RedisDao.getInstance()
                        .getCacheUtags(joinActions.keySet());
        //找出当前没有用户短期兴趣标签的 deviceIds
        List<String>noShortTagUids = new LinkedList<>();
        for(String uid : joinActions.keySet()){
            if(!shortUtags.containsKey(uid)){
                noShortTagUids.add(uid);
            }
        }
        //批量找出没有短期兴趣标签的所有用户的长期兴趣标签，内容格式：
        //<设备 ID，<标签 ID，相关权重>>，
        //一般为老用户的每天第一次访问时使用
        Map<String,Map<String,Double>>longUtags =
                            MongoDao.getInstance()
                                .getCacheUtags(noShortTagUids);
```

```
//既无短期标签又无长期标签的设备 ID，即新增用户，
//只有首次访问时如此
List<String>noTagUids = new LinkedList<>();
for(String uid : noShortTagUids){
    if(!longUtags.containsKey(uid)){
        noTagUids.add(uid);
    }
}
noShortTagUids.clear();
noShortTagUids = null;
//批量找出所有 recomId 所对应的每次推荐新闻所属标签集，内容
//格式：<recomId,[当前 recomId 推荐的新闻所属标签 ID 集]>
Map<String,List<String>>recomIdTags =
    MongoDao.getInstance().getCacheRtags(recomIdSet);
recomIdSet.clear();
shortUtags = null;
//批量找出所有用户参与的新闻所属标签，
//内容格式：<新闻 ID，<标签 ID，相关权重>>
Map<String,Map<String,Double>>newsIdTags =
    EsDao.getInstance().getCacheNtags(newsIdSet);
newsIdSet.clear();
newsIdSet = null;

//每天非首次访问的老用户短期兴趣标签更新
if(shortUtags!=null &&shortUtags.size()>0){
    this.statsUtags(joinActions, shortUtags,
                recomIdTags, newsIdTags, userLbs);
}
//每天第一次访问的老用户，短期兴趣标签更新
if(longUtags!=null &&longUtags.size()>0){
    this.statsUtags(joinActions, longUtags,
                recomIdTags, newsIdTags, userLbs);
}
//新增用户的首次访问，短期兴趣标签更新
if(noTagUids.size()>0){
    this.statsNewUtags(joinActions, noTagUids,
                    recomIdTags, newsIdTags, userLbs);
}
//返回值为最终计算出的含有用户标签的用户个数
rtNum = userLbs.size();
if(userLbs.size()>0){
    //批量存入 Redis
    RedisDao.getInstance().addShortUtags(userLbs);
}
//及时回收垃圾
joinActions.clear();
joinActions = null;
recomIdTags.clear();
recomIdTags = null;
newsIdTags.clear();
newsIdTags = null;
```

```
            longUtags.clear();
            longUtags = null;
            shortUtags.clear();
            shortUtags = null;
        } else {
            int middle = this.uActions.size() / 2;
            StatsTask left = new StatsTask(
                            uActions.subList(0, middle) );
            StatsTask right = new StatsTask(
                uActions.subList( middle, this.uActions.size() ) );
            left.fork();
            int leftNum = left.join();
            int rightNum = right.compute();
            rtNum = leftNum + rightNum;
        }
        return rtNum;
    }

    //计算当下这一小批用户的短期兴趣标签权重并更新，并存到参数：userLbs
    public void statsUtags(
          Map<String,Map<String,Map<String, String>>> joinActions,
                    Map<String,Map<String,Double>> uTags,
                    Map<String,List<String>> recomIdTags,
                    Map<String,Map<String,Double>> newsIdTags,
                            Map<String,String>userLbs){
        Map<String,Map<String,String>>tmpActions = null;
        Set<String>hisRecomTids = null;
        Map<String,Double>curUserNtags = null, tmpTags=null;
        List<String>tmpRids = null;
        Map<String,Double>tmpUtags = null,tmpNewUtags = null;;
        //标签命中率：用户近段时间所参与的新闻对应所有不重复标签个数
        // 除以为其推荐的不重复标签总数
        double tagHitRatio = 0;
        double tmpTwt = 0;
        for(Entry<String,Map<String,Double>> item
                                        : uTags.entrySet()){
            //计算钱用户标签权重
            tmpUtags = item.getValue();
            tmpNewUtags = new HashMap<String,Double>();
            //获取每个用户最新一段时间的用户行为，
            //内容格式：<行为类型，<新闻 ID：recomId>>
            tmpActions  = joinActions.remove(item.getKey());
            if(tmpActions!=null){
            //用户当前最近一个计算周期内，所有参与新闻对应的标签及权重
            curUserNtags = new HashMap<String,Double>();
                //用户当前最近一个计算周期内，所有参与新闻对应的 recomId 集
            hisRecomTids = Sets.newHashSet();
            for(Map<String,String>nrMap : tmpActions.values()){
                for(Entry<String,String>nrItem
                                            : nrMap.entrySet()){
                    tmpTags = newsIdTags.get(nrItem.getKey());
                    if(tmpTags!=null){
```

```
        for(Entry<String,Double>ntw
                        : tmpTags.entrySet()){
            if(curUserNtags
                .containsKey (ntw.getKey())){
                curUserNtags.put  (ntw.getKey(),
                    ntw.getValue()
               +curUserNtags.get(ntw.getKey()));
                }else{
                   curUserNtags.put(ntw.getKey(),
                   ntw.getValue());
                }
            }
            tmpRids = recomIdTags
                    .remove(nrItem.getValue());
            if(tmpRids!=null){
                hisRecomTids.addAll(tmpRids);
            }
        }
    }//for
}//for
//该用户标签命中率
tagHitRatio = 1.0 * hisRecomTids.size()
                        / curUserNtags.size();
double pExpv = Math.exp(-1 * tagHitRatio);
double nExpv = Math.exp(-1 * (1-tagHitRatio));
for(String tid : hisRecomTids){
    //正反馈标签权重计算
    if(curUserNtags.containsKey(tid)){
    tmpTwt = curUserNtags.get(tid) * (1/(1+pExpv));
        if(tmpUtags.containsKey(tid)){
            tmpNewUtags.put(tid,
                tmpTwt + tmpUtags.remove(tid));
        }else{
            tmpNewUtags.put(tid, 1/(1 + pExpv));
        }
    }else{//负反馈标签权重计算
        if(tmpUtags.containsKey(tid)){
            tmpTwt = tmpUtags.remove(tid)
                        * (1/(1+nExpv));
            tmpNewUtags.put(tid, tmpTwt);
        }else{
            tmpNewUtags.put(tid, 1/(1+nExpv) - 1);
        }
    }
}
if(pExpv<this.redutionRatio){
    pExpv = this.redutionRatio;
}
//当前用户历史兴趣标签权重衰减，参考牛顿冷却定理
for(Entry<String,Double>oldTwt
                    : tmpUtags.entrySet()){
    tmpTwt = Math.exp(-1 * pExpv)
```

```
                        * oldTwt.getValue();
                    tmpNewUtags.put(oldTwt.getKey(), tmpTwt);
                }
            }//if(tmpActions!=null)

            //标签权重归一化
            if(tmpNewUtags.size()>0){
                double totalWt = 0;
                for(double wt : tmpNewUtags.values()){
                    totalWt += wt;
                }
                JSONObjectrtJson = new JSONObject();
                for(Entry<String,Double>subItem
                            : tmpNewUtags. entrySet()){
                    rtJson.put(subItem.getKey(),
                            subItem.getValue()/totalWt);
                }
                userLbs.put(item.getKey(), rtJson.toString());
                tmpNewUtags.clear();
                tmpNewUtags = null;
            }
        }//for
    }

    //计算当下新用户首次访问时的短期兴趣标签权重，并存到参数：userLbs
    public void statsNewUtags(
          Map<String,Map<String,Map<String, String>>> joinActions,
                        List<String> noTagUids,
                        Map<String,List<String>> recomIdTags,
                    Map<String,Map<String,Double>> newsIdTags,
                        Map<String,String>userLbs){
        Map<String,Map<String,String>>tmpActions = null;
        Set<String>hisRecomTids = null;
        Map<String,Double>curUserNtags = null, tmpTags=null;
        List<String>tmpRids = null;
        Map<String,Double>tmpNewUtags = null;;
        //标签命中率：用户近段时间所参与的新闻对应所有不重复标签个数
        //除以为其推荐的不重复标签总数
        double tagHitRatio = 0;
        double tmpTwt = 0;
        for(String newUid : noTagUids){
            tmpNewUtags = new HashMap<String,Double>();
            //获取用户计算周期内的用户行为，内容格式：
            // <行为类型，<新闻 ID：recomId>>
            tmpActions  = joinActions.remove(newUid);
            if(tmpActions!=null){
                //当下用户当前最近一个计算周期所有参与新闻对应标签及权重
                curUserNtags = new HashMap<String,Double>();
                //当下用户当前最近一个计算周期所有参与新闻对应 recomId 集
                hisRecomTids = Sets.newHashSet();
                for(Map<String,String>nrMap : tmpActions.values()){
                    for(Entry<String,String>nrItem
```

```
                        : nrMap.entrySet()){
                tmpTags = newsIdTags.get(nrItem.getKey());
                if(tmpTags!=null){
                    for(Entry<String,Double>ntw
                            : tmpTags.entrySet()){
                        if(curUserNtags
                            .containsKey (ntw.getKey())){
                                curUserNtags.put(
                                    ntw.getKey(),
                                    ntw.getValue()
                            + curUserNtags.get(ntw.getKey()));
                        }else{
                            curUserNtags.put(ntw.getKey(),
                             ntw.getValue());
                        }
                    }
                    tmpRids = recomIdTags
                            .remove(nrItem.getValue());
                    if(tmpRids!=null){
                        hisRecomTids.addAll(tmpRids);
                    }
                }
            }//for
        }//for
        //该用户标签命中率
        tagHitRatio = 1.0 * hisRecomTids.size()
                        / curUserNtags.size();
        double pExpv = Math.exp(-1 * tagHitRatio);
        double nExpv = Math.exp(-1 * (1-tagHitRatio));
        for(String tid : hisRecomTids){
            //正反馈标签权重计算
            if(curUserNtags.containsKey(tid)){
                tmpTwt = curUserNtags.get(tid)
                        * (1/(1+pExpv));
                tmpNewUtags.put(tid, 1/(1 + pExpv));
            }else{//负反馈标签权重计算
                tmpNewUtags.put(tid, 1/(1+nExpv) - 1);
            }
        }
    }//if(tmpActions!=null)
    //标签权重归一化
    if(tmpNewUtags.size()>0){
        double totalWt = 0;
        for(double wt : tmpNewUtags.values()){
            totalWt += wt;
        }
        JSONObjectrtJson = new JSONObject();
        for(Entry<String,Double>subItem
                    : tmpNewUtags. entrySet()){
            rtJson.put(subItem.getKey(),
                    subItem.getValue()/totalWt);
        }
```

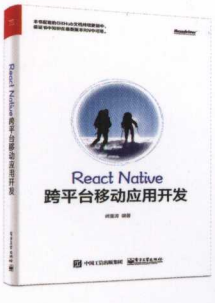
```
                    userLbs.put(newUid, rtJson.toString());
                    tmpNewUtags.clear();
                    tmpNewUtags = null;
                }
                }//for
            }
        }
```

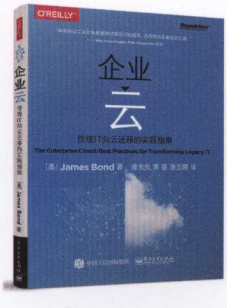
有了此主程序，参考 Light_drtc 中 JN 启动说明，就可以完成整个用户标签的实时计算更新。

9.4 总结

本章主要从当前互联网行业的个性化推荐系统工程应用角度，向读者朋友介绍每个模块组成，并对其做了详细说明。然后以新闻推荐系统中的用户画像实时更新为实例，将系统完整架构、相关数据字典和系统更新流程为例，向读者进一步说明如何使用本书所提倡的分布式实时计算框架 Light_drtc 处理问题。

好书力荐





拒绝堆砌臃肿，支持纯正原创

出版事宜请关注 新浪微博 weibo.com @ 半亩方塘 _

投稿邮箱：sxy@phei.com.cn

加入本书读者 QQ 群 465398813，以书会友，资源共享！

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

专家力荐

本书是作者基于自己多年的思考和实践经验，经过不断提炼而得的。一方面把现在业界实时计算领域常用的产品和技术进行深入介绍，让读者对该领域中的各个产品如何使用不再迷茫，也为具有一定经验的从业者对下一步系统中关键服务如何选型和优化提供了新的方向；另一方面作者博采众家之长而创造的light_drtc，降低了中小企业在分布式实时计算领域的门槛，使大数据处理能力触手可得。最后通过作者的实例，在了解大数据技术如何应用的同时，也能了解到在实例背后作者所体现出的思路、方法和方案。希望通过本书能让更多的人了解大数据处理，有更多的企业挖掘出自身的数据价值。

中国建筑电商CTO　邓威

成光对于架构的不断钻研和踏实肯干给我留下了很深的印象，非常有幸见证了他这套分布式实时计算系统在新闻推荐领域中的应用。这本书结合他多年的一线实践经验，详细阐述了分布式计算系统、搜索架构和数据库等在企业应用的经历，对于初学者和想深入理解这一方面知识的同学会起到很好的引导。感谢成光为国内的工程架构贡献自己的一份力量！

腾讯技术副总监　鞠奇

分布式实时计算技术对于大数据技术来讲非常重要，不过在技术上能够讲解得非常透彻的中文技术书籍并不多。认识王成光时他正沉浸在开发light_drtc的状态中，关于分布式实时处理技术，我们聊了很多，有很多技术的见解也很一致。令人非常高兴的是，他不仅将自己实践积累的产品light_drtc开源出来，同时将自己多年技术积累的经验以写书的形式奉献给了广大对分布式实时计算技术有兴趣的技术人员。《分布式实时计算框架原理与实践案例》不止是一本对当前实时流计算技术进行解析的书，同时也是作者对在工作中实战经验总结的一本书。授人以鱼不如授人以渔，我相信这本书能够给大数据技术人员，尤其是对大数据流处理技术有兴趣的人带来很大的帮助。

TalkingData研发副总裁　阎志涛

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：孙学瑛
封面设计：李　玲

上架建议:计算机>架构设计

ISBN 978-7-121-29620-8
9 787121 296208
定价：79.00元